Ahmad Naser eddin

Large Scale Parallel Subgraph Search

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2016

Ahmad Naser eddin

Large Scale Parallel Subgraph Search

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Advisor: Prof. Pedro Ribeiro

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2016

Para o meu país e para a minha familia

Acknowledgments

Foremost, I would like to express the deepest gratitude and appreciation to my supervisor, Prof. Pedro Ribeiro. Without his guidance, persistent help and incredible patience this dissertation would not have been possible. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it.

I would like to thank the Erasmus Mundus program especially Phoenix project, for the financial support that was allowed to me during my master degree. Also I had a grant from CRACS & INESC-TEC and I want to thank them for it.

I want to thank all the helpful people who I met during these two years in the university of Porto. In particular David Aparício who was always willing to help and give his best suggestions, and Miguel Araújo who gave me the chance to use a different computational infrastructure. I am grateful to Lama Alimam, for her continuous support and motivation, she has been like my elder sister in Portugal.

Finally, I must express my very profound gratitude to my parents for providing me with unconditional love, unfailing support and continuous encouragement throughout my years of study.

Abstract

Networks are powerful in representing a wide variety of systems in many fields of study, from computer science and mathematics to biology and chemistry. In the late 1990s when researchers started to intensively study diverse real-world networks and their models, the term *complex networks* came to use. Networks are composed of smaller substructures (subgraphs) that characterize them and give important information related to their topology and functionality. Therefore, discovering those patterns is very important towards mining the features of networks. Algorithmically, subgraph search in a network is a *computationally hard* problem and the needed execution time grows exponentially as the size of the subgraph or the network increases.

The main goal of this thesis is to scale up the execution of subgraph counting, increasing the limits of its applicability. For that purpose, we present a dynamic iterative parallel MapReduce strategy general enough to be applicable to any *tree-like* unbalanced search. MapReduce is pervasive in every computing cloud, giving our work a broad scope.

We applied our strategy to two different state-of-the-art algorithms: *G-Tries* and *ESU*. We evaluated our implementions in a large set of representative complex networks from different fields. The results obtained are very promising. We achieved an efficiency of up to 92.5% using 32 independent cores and a top speedup of 45.5 when using 64 cores that share part of their cache. On average, our implementation is able to consistently obtain an almost linear speedup up to 32 cores, with an average efficiency larger than 80%.

In order to make it easier for the users to do subgraph counting in a fast and user friendly way, we developed an initial version of a plugin for the well known *Cytoscape* network analysis platform. The prototype has a fully functional sequential version of *G-Tries* algorithm and it is already available on the app market.

To sum up, this work expands the applicability of subgraph counting algorithms, paving the way for new discoveries and insights in network science.

Resumo

As *redes* são uma poderosa maneira de representar uma grande variedade de sistemas em muitas áreas de estudo, desde a ciência de computadores e a matemática, até à biologia ou química. No final da década de 90, quando os investigadores começaram a estudar mais intensivamente toda uma série de modelos de redes do mundo real, o termo *redes complexas* começou a ser usado. As redes são compostas por pequenas subestruturas (subgrafos) que podem caracterizá-las e que dão importante informação sobre a sua topologia e funcionalidade. Por isso mesmo, descobrir estes padrões é muito importante no que toca a analisar as propriedades das redes. Do ponto de vista algorítmico, a procura de subgrafos é uma tarefa *computacionalmente difícil* e o tempo de execução necessário cresce exponencialmente à medida que o tamanho dos subgrafos ou das redes aumenta.

O principal objectivo desta tese é melherar a escalabilidade da contagem de subgrafos, aumentando assim os limites da sua aplicabilidade. Com este propósito em mente, apresentamos aqui uma estratégia paralela baseada no conceito de *MapReduce* usando iterações dinâmicas e sendo geral o suficiente para ser aplicável em qualquer pesquisa não balanceada em forma de árvore. Plataformas MapReduce são ubíquas em *clouds* de computação, dando ao nosso trabalho um grande alcance.

Aplicamos a nossa estratégia em dois diferentes algoritmos que são o estado-da-arte: *G-Tries* e *ESU*. Avaliamos as nossas implementações num leque variado de redes complexas representativas de vários domínios. Os resultados obtidos são muito prometedores. Alcançamos uma eficiência até 92.5% usando 32 cprocessadores independentes e um máximo de 45.5x de *speedup* quando usando 64 processadores que partilham parte da sua cache. Em média, a nossa implementação consegue de forma consistente obter um *speedup* quase linear até 32 processaores, com uma eficiência média maior do que 80%.

Para que seja mais fácil a um utilizador calcular a frequência de subgrafos de uma maneira mais rápida e amigável, desenvolvemos uma versão inicial de um *plugin* para o *Cytoscape*, uma plataforma de análise de redes. O protótipo desenvolvido contém

uma versão completamente funcional da pesquisa sequencial usando *G-Tries* e está já disponível para download no ecossistema de *plugins* do Cytoscape.

Em suma, este trabalho expande a aplicabilidade dos algoritmos de contagem de subgrafos, abrindo o caminho para novas descobertas na análise de redes.

Contents

List of Tables

List of Figures

List of Algorithms

Introduction 1

Complex networks describe a wide range of systems in nature and society. These networks can appear naturally, such as protein interaction networks or food chains, or they can be man-made, like the internet or the power grid. Complex networks form the backbones of complex systems and their analysis is of vital importance in a multitude of fields. Given their scale, this task requires a lot of computation power. However, with the recent advances in technology, analyzing complex networks has become possible in much more detail than what was previously possible [Erc15].

1.1 Motivation

In recent years network science has emerged as an important multidisciplinary field, with applications on areas such as computer science, physics, biology or engineering. Although its roots are on the older field of graph theory, the analysis of networks has been receiving an increasing attention due mainly to two important factors [Bar16].

The first contributing factor is the availability of network maps, due to technological advances that have provided an enormous amount of data which can be represented by networks. These can be social (e.g. facebook [ML12], twitter [DDLMM13] or co-authorship networks [New06]), biological (e.g. brain [BS12], protein interaction [AA04] or metabolic networks [DA05]), spatial (e.g. flights [ARF16], roads [ŠB11] or power grids [WS98]), textual (e.g. semantic networks [BMZ02]) or even software networks (e.g. module dependency [vB12]). Several public network repositories are nowadays available [Kun16, LK14]. Figure 1.1 givens an example visualization of a real network, in this case the worldwide air traffic between airports.

The second contributing factor is the realization that complex networks from different areas share non-trivial common topological features. This universality of characteristics serves as a guiding principle for network analysis and gives a wide applicability to any discoveries. Two examples of common properties are power law degree distributions (leading to *scale free* networks [Bar09]) and small average distances between

Figure 1.1: Visualization of the air traffic network, taken from [Gra16].

nodes (leading to *small world* networks [WS98]).

In order to extract information from networks, practitioners have have a wide range of measurements available [CRTVB07]. Some of them describe properties at the node level (such as its degree) while others describe global metrics (such as the average distance between nodes). One other way of analyzing a network is to use an intermediate approach, looking at small topological patterns of interconnections, bigger than a single node but smaller than an entire network, and trying to understand what is their role on the global behavior of the network. These small substructures are subgraphs and they can be seen as basic building blocks of networks [MSOI+02]. Figure 1.2 illustrates the occurrences of a small *triangle* pattern in a network.

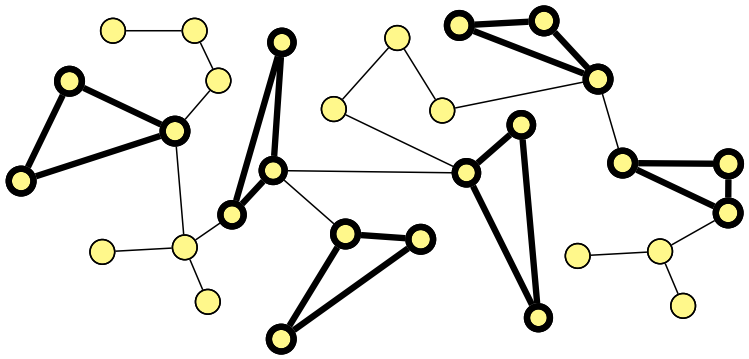

Figure 1.2: An example of a pattern of connections in a network. In this case

Subgraphs have been widely used to uncover the design principles of complex networks networks. Two very important related concepts are *network motifs* and *graphlets*. A *network motif* is a small connected subgraph that appears in significantly higher numbers than what one would expect. This idea of overrepresented subgraphs was introduced by Milo et al. [MSOI+02], and a more detailed definition can be seen in in Section 2.3). The notion of *graphlets* is similar, but it essentially disregards overrepresentation and instead focuses on the number of times a node appears in a certain position of subgraph [Pržž07]. For instance, it quite different to be at the center or at the periphery of a star shaped subgraph.

Both of these concepts have at its core the task of counting the occurrences of a given set of subgraphs, that is, computing a *subgraph census*. This is however a challenge, since algorithmically speaking it is a **computationally hard** problem. In fact, just knowing if a subgraph appears at all in another larger graph is an NP-complete problem [Coo71], and finding the exact number of times it appears is an even harder task. Given this, the needed execution time grows exponentially as we increase the size of the network or the size of the subgraphs being searched.

Performance is the main factor that users care about. One way of improving the performance was getting faster processors, but nowadays the individual processors are not getting much faster and the devices are just getting more processors. Parallelism appears therefore naturally as one the current most efficient ways for improving performance [Boy08], and it implies splitting the work between those processors or between the cores inside one processor.

One way of introducing parallelism on an algorithm is to use the MapReduce framework, which is a programming model that automatically parallelizes the computation across large clusters, handles machine failures and makes efficient use of the network and disks by scheduling the communication between workers (cpus, cores, threads, etc) [DG08]. MapReduce was initially developed by Google in 2008 and it has since been widely there. For instance, Dean and Ghemawat estimated that an average of one hundred thousand MapReduce jobs were being executed daily on Google's clusters [DG08]. There are many frameworks that support MapReduce, the most famous one being Hadoop [Apa16a]. However, Hadoop has some drawbacks that will be illustrated in more detail in section 4.1.4.1. A recent novel framework is Spark [ZCF+10]. It supports MapReduce applications retaining MapReduces its two main goals: scalability and fault tolerance. Spark is more than 10 times faster than Hadoop on disk and it can be up to 100 times faster in memory. To achieve the mentioned goals Spark introduces an abstraction called RDD, referring to a resilient

distributed dataset. RDD is a read-only collection of objects, partitioned across a set of workers (machines) and in case a partition was lost it can be rebuilt.

An important advantage of using MapReduce is its availability on cloud computing websites. For instance the user could rent computation time in a cluster in any cloud computing provider (e.g. amazon web services) and run the algorithm using as many processors as needed. So the user does not need to have his own powerful machines to run large graphs. By providing subgraph counting algorithms in a MapReduce framework we could therefore really make them available to a wider and more general audience of practitioners in different fields.

1.2 Goals and Contributions

The main goal of this work is to provide and efficient parallel MapReduce algorithm for **Subgraph Census** and make it available to end users in a friendly way.

Since the used technologies do not provide a tight integration with native programming languages, and given that our case study algorithms (ESU [Wer06] and G-Tries [RS14b]) were originally implemented in C++, one constribution of our work is a java implementation of the sequential versions of these two subgraph census algorithms.

The original ESU uses an isomorphism test after each individual subgraph occurrence, and we improved upon this strategy. Using an idea similar to what was done in FaSE algorithm [PR13], we delay the isomorphism tests to the end of the enumeration process while also identifying that certain occurrences are of the same type, greatly decreasing the amount of needed isomorphism tests.

Another major contribution is a MapReduce Strategy for unbalanced *"tree-like"* parallel search. The strategy works iteratively as follows: first the work is divided in a round robin way between the workers; then, after a specific amount of time (a threshold) all the workers should stop and save their state; afterwards the master node (controller) collects those states, resets the threshold and divides the remaining work into the workers again. The master node keeps iterating the work until there is no more work to do, actively promoting dynamic load balancing by changing the threshold according to the state of computation (numbers of workers and the amount of time each of them potentially was idle during the last iteration).

In this work we include an application of this parallel strategy to both ESU and g-tries algorithms, giving extensive algorithmic details. Furthermore, we have made a comprehensive experimental evaluation of our implementation of both parallel algo-

rithms in a large set of representative networks from different fields. The obtained results are very promising and we achieve close to linear speedups up to 32 cores (average efficiency close to 80%) and a top speedup of 45.5 with 64 cores (limited by our available hardware, because for more than 32 the cores are not 100% independent and share some cache).

The last contribution of our work was the development of an initial subgraph census plugin for Cytoscape software [SMO+03], providing easy and friendly access to some of our algorithms to the end user. Cytoscape is widely used in the Systems Biology community and by having a plugin available in the app market, a biologist could use the software by simply clicking on a few buttons on an already known environment as opposed to having to install a dedicated command line piece of software.

1.3 Organization

This thesis is structured in seven major chapters:

Chapter 1 - Introduction. Offers an overall view of the problem being studied in this thesis as well as the motivation behind it. Also it shows the goals and contributions and it additionally presents the thesis organization.

Chapter 2 - Subgraph Search. Introduces a common graph terminology that will be used throughout the thesis, gives a detailed description about the computational problem being studied, and gives some potential applications.

Chapter 3 - Algorithms for Subgraph Census. Gives an explanation of the state of art and introduces a detailed description of the two main algorithms being used in this thesis. It also explains the already existing parallel approaches.

Chapter 4 - Parallel Approaches. Justifies the option to apply iterative MapReduce in parallelizing tree-like algorithms in finding motifs and explains the approach with details for the two algorithms (ESU and G-Tries) which are used as case studies.

Chapter 5 - Performance Evaluation. We access the scalability of our implementations by doing a thorough study using a large set of representative networks from different fields.

Chapter 6 - Cytoscape Plugin. Introduces our plugin and details the technologies being used.

Chapter 7 - Conclusions and Future Work. Concludes the thesis, by summarizing the contributions made and giving directions for possible future work.

Subgraph Search 2

In this chapter we introduce a common graph terminology to be used throughout the thesis, we formally define the problem we are tackling and we describe some of its applications.

2.1 Graph Terminology

Networks are modeled with the mathematical object *graph*. A graph G is composed of a collection of vertices (or nodes) $V(G)$ and a set of edges (or connections) $E(G)$. The *size* of the graph is the number of vertices it has, and it is written as $|V(G)|$. The term *k-graph* refers to a graph of size k. Edges are composed of pairs of vertices (u, v) where u and v are from the set of vertices. The order of the pair matters only in the *directed* graphs, and in that case the pair will be $(from, to)$ expressing the direction of the edge, while in *undirected* graphs edges have no direction as its name implies. In undirected graphs the *degree* of a vertex is the number of edges it has. However, in directed graphs the vertex has two degrees, the *indegree* (number of ingoing edges) and the *outdegree* (number of outgoing edges).

If the graph has no self-loops or multiple edges then it is considered *simple graph*. *Self-loops* are edges from the vertex to itself. The expression *multiple edges* implies that there might exist two or more edges connecting the same pair of nodes. In this thesis we assume that we are only dealing with simple graphs.

The *neighbourhood* of a *vertex* u is the set of vertices that share an edge with u. More formally, $u \in V(G), v \in V(G) \Rightarrow N(u) = v : (u, v) \in E(G)$.

Vertices are distinguished by assigning them *labels* from 0 to $|V(G) - 1|$, thus the comparison $u < v$ refers to a comparison between their labels and in this case it means that the vertex u has a lower label than v. These labels are used by both g-Tries and ESU as part of their symmetry breaking conditions, which allow these algorithms to only count once each subgraph occurrence (more details in section 3.2).

A binary adjacency matrix G_{Adj} is used to represent the graph G, each cell in the adjacency matrix $G_{Adj}[u][v]$ represents the existence of an edge (u, v), when $(u, v) \in E(G) \Rightarrow G_{Adh}[u][v] = 1$; otherwise $G_{Adj}[u][v] = 0$. Figure 2.1 shows an example of an adjacency matrix of a directed 5-graph.

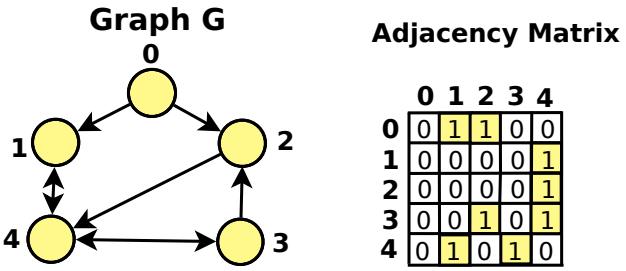

Figure 2.1: Adjacency matrix for an example graph G.

Some graphs contain another graphs. The contained graph is called a *subgraph*. A graph H is a *subgraph* of a graph G if $V(H) \in V(G)$ and $E(H) \in E(G)$. This subgraph H is called *induced* if $\forall u, v \in V(G), (u, v) \in E(H)$ if and only if $(u, v) \in E(G)$, and if the graph G has a set of nodes that induce H then this set is called an *occurrence* or a *match*. Distinct matches must have at least one different vertex. The number of occurrences of H in G is called its *frequency*. Figure 2.2 shows an example of a graph G, a subgraph H and its four occurrences.

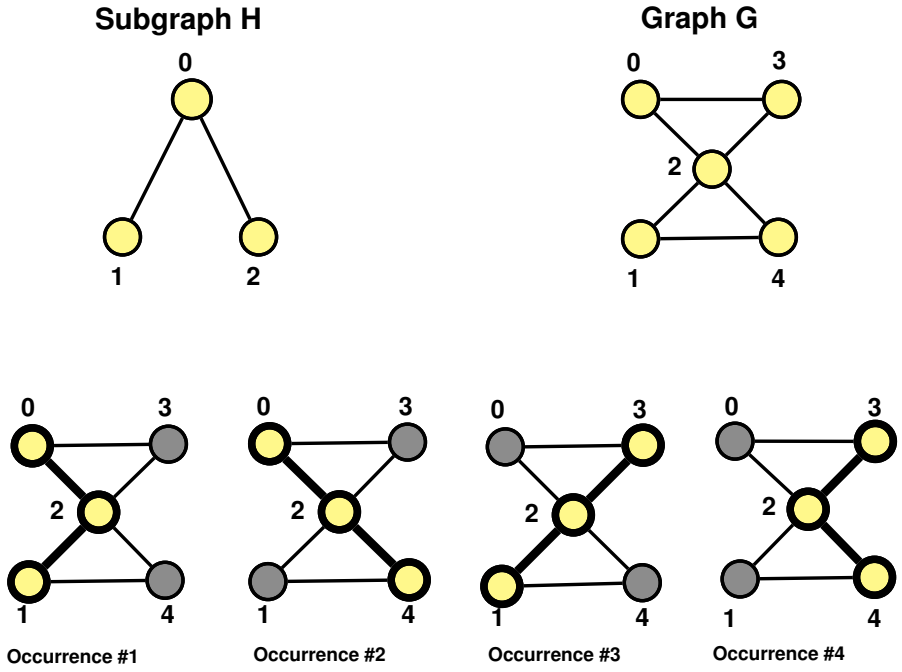

Figure 2.2: Induced occurrences of a subgraph H in a larger graph G.

The *neighborhood* of a *subgraph* H is the union of all the neighborhoods of its vertices, $N(H) =$ the union of $N(v) \forall v \in V(H)$.

Two graphs G_1 and G_2 are *isomorphic* ($G1 \sim G2$) if there is one to one mapping between their vertices and there is an edge in G_1 if and only if there is an edge between the corresponding vertices in G_2. This problem is computationally *hard* and it is neither known to be solvable in polynomial time nor NP-complete [MP14]. Another similar but different problem is *subgraph isomorphism*, in which given two graphs G_1 and G_2 we need to determine if G_1 contains a subgraph which is isomorphic to G_2. This problem is known to be NP-complete [Erc15].

2.2 Subgraph Census Problem

As the last section described, just knowing if a graph appears as a subgraph of another larger graph is already an NP-complete problem. The main computational problem being solved in this thesis is an even more general version of this problem, that is, to actually compute the number of occurrences of each subgraph type. Our goal is precisely to improve the efficiency and scalability of algorithms for this task an we now define more formally the problem we are tackling.

Definition 2.1 (**Subgraph Census Problem**). *Given a graph G and a subgraph size k, determine the exact frequencies of all induced occurrences of all possible k-subgraph types in G. Two occurrences are considered distinct if they have at least one vertex that they do not share. Other vertices can overlap.*

In some cases we may be interested on a smaller set of subgraphs than the entire set of size k, but the most common application is really to count all subgraphs of a given size. Figure 2.3 shows an example of a network with six induced occurrences of 3-subgraphs.

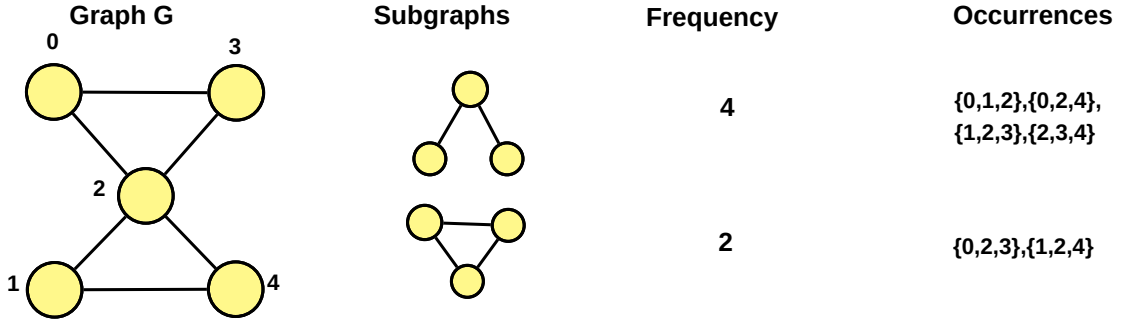

Figure 2.3: Occurrences of subgraphs of size 3 in a graph G.

The number of possible subgraphs of a certain size k differs between directed and undirected graphs. Figure 2.4 displays all the possible subgraph types of size 3.

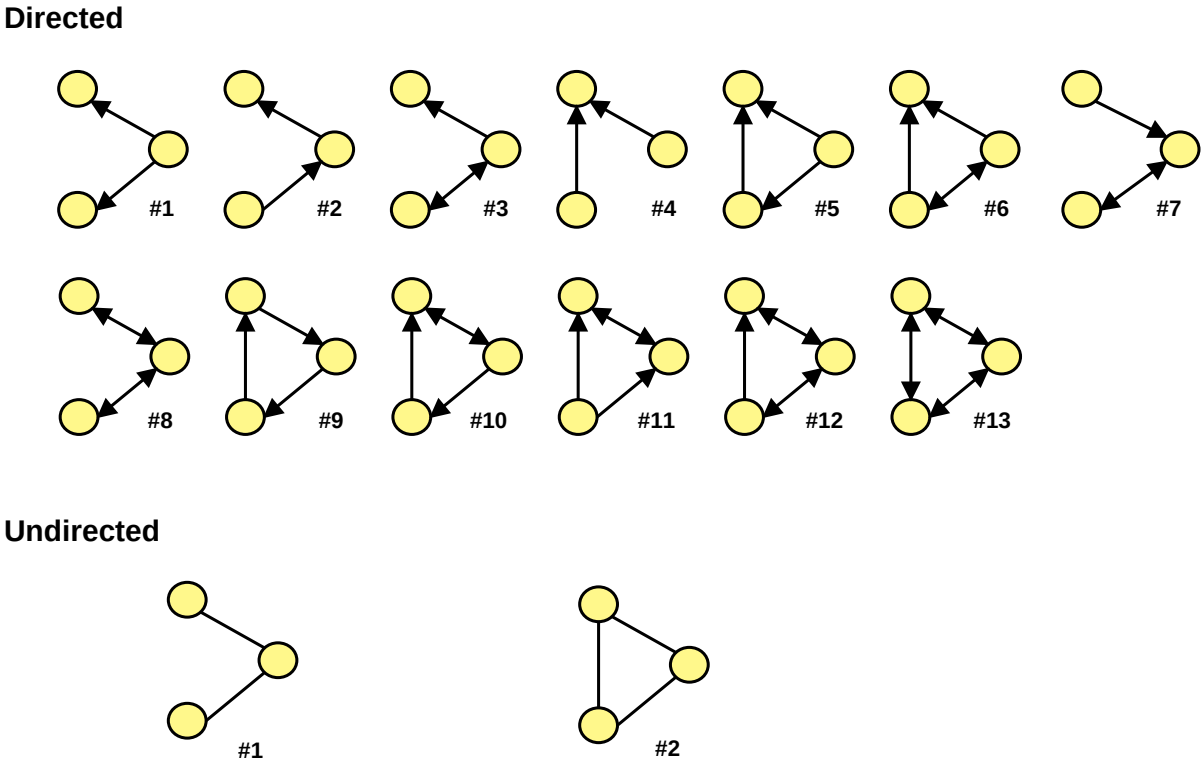

Figure 2.4: All possible directed and undirected subgraph types of size 3.

It is clear that for the same number of vertices, the number of possible directed graph types grows faster than what happens in the undirected case. Table 2.1 expresses this difference up to size 10. These numbers have an huge impact on the tractability of the subgraph census problem, since as you can see, the number of possible types grows exponentially as k increases.

Size k	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Number of Undirected Subgraphs	**2**	**6**	**21**	**112**	**853**	**$\approx 10^4$**	**$\approx 10^5$**	**$\approx 10^7$**
Number of Directed Subgraphs	**13**	**199**	**9364**	**$\approx 10^6$**	**$\approx 10^9$**	**$\approx 10^{12}$**	**$\approx 10^{16}$**	**$\approx 10^{20}$**

Table 2.1: Number of possible directed and undirected subgraph types with k vertices up to size 10.

2.3 Subgraph Census Applications

Subgraph census is an essential core computational task used in many network analysis methodologies. One very important and widely used methodology involves the concept of *network motifs*, a term introduced by Milto et al. in 2002 [MSOI+02]. A motif can be seen as a connected small subgraph that appears in significantly higher number than what one would expect. More formally, it is an induced subgraph that appears more frequently in the original network than in similar randomized networks. The definition of *similarity* may differ if we want a more specific null model, but the most classic model for the randomized networks is to keep the same number of nodes, number of edges and to preserve the degree sequence. An example of this class of similar random networks is shown in figure 2.5.

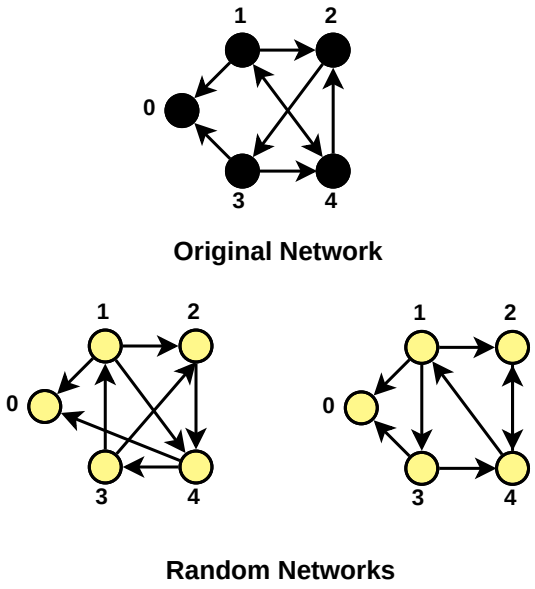

Figure 2.5: Example of similar random networks.

Motifs provide a very general and flexible characterization, helping in gaining a deeper understanding of the topological features of the networks being analyzed. Its first application was to classify networks based on the types of motifs [MSOI+02]. Since then they have been widely used in many different fields such as biology ([AA04, SOMMA02, SK04, KMP+01]) or sociology ([CRBS12, KFMH+11]).

To better exemplify its usage we will now give some practical use cases. Computer programs have a textual representation following syntactic rules dictated by a programming language. Each program is divided into smaller software entities which are given unique names. Those entities can have relations between each other. In object oriented programming this relation can be inheritance, or it could be simpler like if a

class defines an instance of the other. It is useful to represent those programs using graphs, where vertices represent classes and edges represent the relation between them. Motifs were used to detect the patterns in software class diagrams and indicate static dependencies between classes [VS05]. Figure 2.6 shows some examples of network motifs of size 4 found in software graphs. Each node in the figure represents a class and the edges indicate the dependencies between classes.

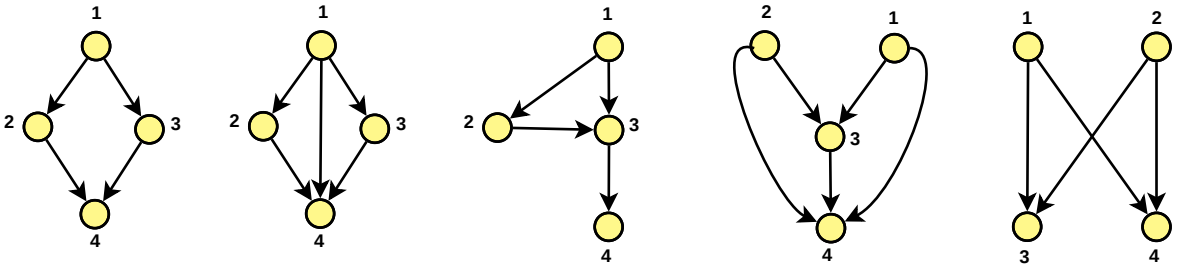

Figure 2.6: Examples of network motifs with size=4 found in software graphs. Adapted from [VS05]

Network motifs were also the basis of a Web Appearance Disambiguation system (WAD) [YCLH06]. WAD uses the hyperlink structure between web pages to have a better search for a particular person. It clusters searched pages using network motifs as an evidence of close relationship between pages. Thus, the WAD system does not require any background knowledge about the users to do the clustering unlike previous systems in the area. Pages were represented as vertices and the hyperlinks as edges. Because the number of incoming links is much smaller than the number of outgoing links and in order to balance the augmented dataset and collect more incoming links they collected the pages that have hyperlinks to the domain name using a host-based augmented process as shown in figure 2.7.

Different variations on the concept of motifs may exist. For instance, *colored motifs* are an extension of the motif concept, where the color of the vertices and edges need to be checked in addition to their connections[LFS06, FFHV07, RS14a]. Another extension uses the *weight* of the connections in order to better understand the network [OSKK05, CRS12]. A *trend motif* is a graph mining approach used for analysis of **dynamic** complex networks where the attributes attached with the vertices or edges change with time [JMA07]. Finally, *anti-motifs* refer to the **under-represented** subgraphs[MIK$^+$04], that occur in lower numbers than what would be expected. Anti-motifs are meaningful for some applications [BGP07].

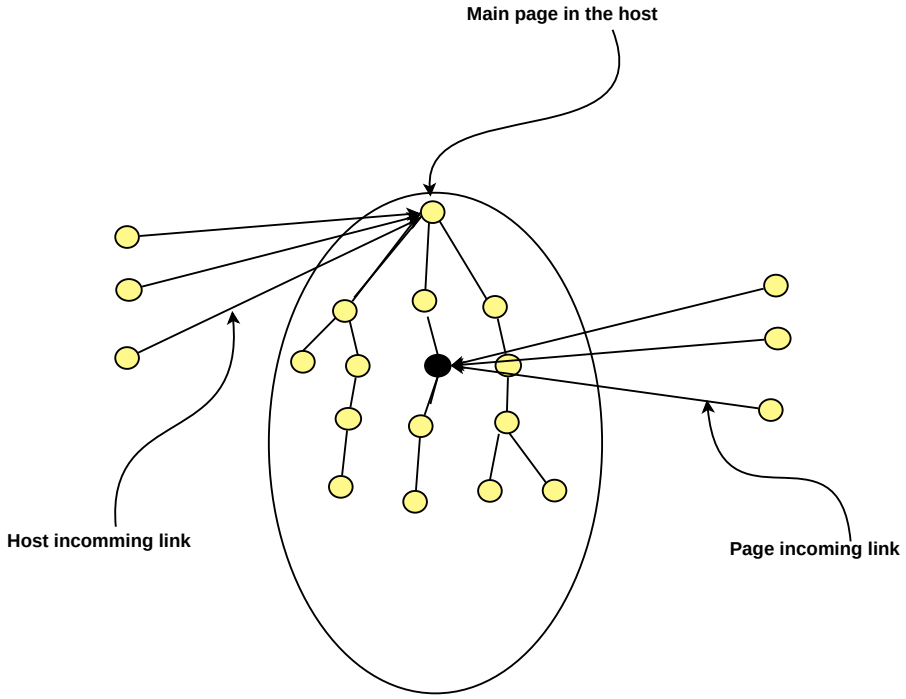

Figure 2.7: The host incoming link. Adapted from [YCLH06]

Algorithms for Subgraph Census

3

The purpose of this chapter is to provide a brief description about the state of art in algorithms for subgraph census and then explain in details the two case studies algorithms that are used throughout this thesis (ESU and G-Tries). Finally we overview the existing parallel approaches.

3.1 Algorithmic Approaches

As seen on the previous chapter, subgraph census is a computationally hard problem closely related to subgraph isomorphism, an NP-complete problem [Coo71]. There have been many possible algorithmics approaches to this problem, that we now describe in more detail.

3.1.1 Historical overview

The first algorithm for motif discovery, called **Mfinder**, is a simple sequential backtracking algorithm [MSOI+02]. The first improvement for this algorithm is **Kashtan** [KIMA04] that was developed in 2004 and it had the possibility of sampling subgraphs. In the same year a different frequency concept was introduced and that led to the implementation of **FPF** [SS04].

The first specialized algorithm with symmetry breaking condition appeared in 2005 and it was named (**ESU** [Wer06]). In 2006 **NeMoFinder** algorithm appeared [CHLN06], and it was the first time that network motifs up to size-12 were discovered. However, NeMoFinder succeeded in doing so by twisting the motif definition and also by looking only for a subset of all possible candidates. In 2007 **Grochow** algorithm [GK07] introduced the ability to search for a single individual subgraph type based on new symmetry breaking technique. **Kavosh** [KAE+09] and **MODA** [OSMN09] are algorithms that appeared in 2009 with a behavior closer to ESU and Grochow. In 2010 the **G-Tries** algorithm [RS10] appeared and outperformed all the previous approaches. It

depends on a novel specialized data-structure encapsulating common substructure. An improvement to the ESU style class of algorithms appeared in 2013 and it is called **FaSE** [PR13]. It avoids the need of an isomorphism test for each subgraph occurrence by using a customized tree data structure. The last two algorithms, G-Tries and FaSE, are the state-of-the-art in what concerns general subgraph census. For specific sets of subgraphs, combinatorical approaches such as ORCA [HD14] or acc-Motif [MMFDC14] exist, but they limit the expressability of the algorithm to a limited pre-defined set of subgraph types.

3.1.2 Strategy types

Sequential subgraph counting algorithms can be divided into three conceptual approaches: **Network-centric, subgraph-centric and set-centric**.

Network-centric methods search for all subgraphs of size k in the target network and then apply isomorphism to determine the type of each subgraph in the occurrences. Examples of this class of algorithms areESU [Wer06], Kavosh [KAE+09] and FaSE [PR13]. These methods can compute the frequency of subgraphs (motifs) which are not in the original network because they compute the complete census of the respective network, and that network could be the original or similar random one, and the random network may contain more subgraphs than the original.

Subgraph-centric methods search only for one individual subgraph type at a time. One weakness of these algorithms is that they do not take advantage of similarities between subgraphs in different searches. For instance, if the difference between two subgraphs was only one node, the computation is done as if they were totally different. An example algorithm of this type is Grochow and Kellis [GK07].

Set-centric methods search for a customized set of subgraphs. They are conceptually in the middle between the previous two, because they do not search for only one subgraph at a time, and also not necessary for all subgraphs of a specific size. This approach was introduced by Ribeiro and Silva with the usage of the g-trie data structure providing an efficient way of representing general sets of subgraphs [RS10].

Table 3.1 shows the different strategy types, and the algorithms that follow each of them. At the present date, *g-tries* provide the fastest general approach for subgraph census and that is why we decided to base our parallel approach on them. In order to showcase the general applicability of our parallel strategies we also provide a *network-centric* parallel algorithm, as explained in Chapter 4.

Approach	Network-centric	Subgraph-centric	Set-centric
Example algorithms	FaSE, Kavosh, NeMoFinder, ESU, FPF, Kashtan, Mfinder	MODA, Grochow	G-Tries

Table 3.1: Subgraph census approaches and example algorithms for each approach.

3.2 Sequential Algorithms

3.2.1 ESU

ESU is a subgraph enumeration algorithm that follows a network-centric paradigm [Wer06]. It appeared in 2005 and it was faster than the existing algorithms, because it avoids redundant computations using symmetry breaking conditions to find the occurrence only once.

3.2.1.1 Computing subgraph frequencies

ESU method starts with a root node and expands from there. When expanding a vertex, only the nodes that have a label (index) greater than this vertex are allowed to be expanded, and by doing that the algorithm avoids symmetries. Algorithm 3.1 describes ESU in pseudo-code: it takes as input a graph $G(V, E)$ and enumerates the size-k subgraphs in this graph. The algorithm iterates throw all the graph vertices, and starting from each vertex V it establishes an array EXT that has the vertices which are neighbors to V and their labels are larger than the label of V (Line: 3). Then EXTEND function is called to extend this vertex and count all subgraph occurrences that exist under V (line: 4). Each call to the extend function is represented by an edge from the vertex representing the caller to the one representing the callee. The callee vertex is labeled $(V_{subgraph}, EXT)$ and located at depth $|V_{subgraph}|$. In the iterative function EXTEND, in each time it calls itself a new node from the extension array (EXT)is selected for expansion, it is removed from the EXT and its exclusive neighbors are added to the new extension array (EXT2) (line: 9). This way, the ESU algorithm ensures that each subgraph will be enumerated exactly once since the non-exclusive neighbors will be considered in another recursion. An occurrence is found when the depth equals the subgraph size (line: 6). ESU uses an efficient third party algorithm called **nauty** [MP14] for doing the isomorphism test after finding each occurrence

(line: 7), nauty was detailed in section 2.1.1.2.

Algorithm 3.1 ESU: Algorithm for computing the frequency of subgraphs of size k in graph G.

```
1: procedure COUNTSUBGRAPHS(G, k)
2:     for all vertex v of G do
3:         EXT ← u ∈ N(V) : v < u
4:         EXTEND({v}, EXT, v)
5: procedure EXTEND(V_subgraph, EXT, v)
6:     if |V_subgraph| = k then
7:         INCREMENTOCCURENCES(NautyCanonicalLabeling(V_subgraph))
8:     for all vertex v in EXT do
9:         EXT2 = EXT ∪ {u ∈ N_excl(w, V_subgraph) : u > v}
10:        EXTEND(V_subgraph ∪ {w}, EXT2, v )
       return
```

3.2.1.2 ESU search tree

Figure 3.1 illustrates the structure of the ESU-Tree and details how ESU enumerates all size-3 subgraphs in the given example which has 5 nodes. Each node in the tree indicates the parameters passed to the EXTEND function, as follows($|V_{subgraph}|, EXT$).

It is assumed that the nodes in the ESU search tree are ordered according to the same order that the subroutines they represent are called.

ESU search tree has the following properties:

1. If $n1$ is a node in the tree, then for every vertex $v \in EXT(n1)$, $n1$ has child node $n2$ where $v \in SUB(n2)$.

2. For each node n in the tree, and for every vertex $u \in EXT(n)$, we have $u > v$. where v is the smallest-label vertex in$SUB(n)$.

3. Let $n1$ and $n2$ be two nodes in the ESU-Tree which have the same parent node and $n1 < n2$, then $SUB(n1)$ contains one vertex $u1$ which is not contained is $SUB(n2)$ and vice versa. That means for every child node $n3$ whose path to the root contains $n2, u1 \notin SUB(n3)$.

3.2.1.3 Graph Isomorphism Algorithms

As explained, one the the steps needed for ESU algorithm is computing graph isomorphism. For the sake of completeness, we explain some of the possible approaches, given

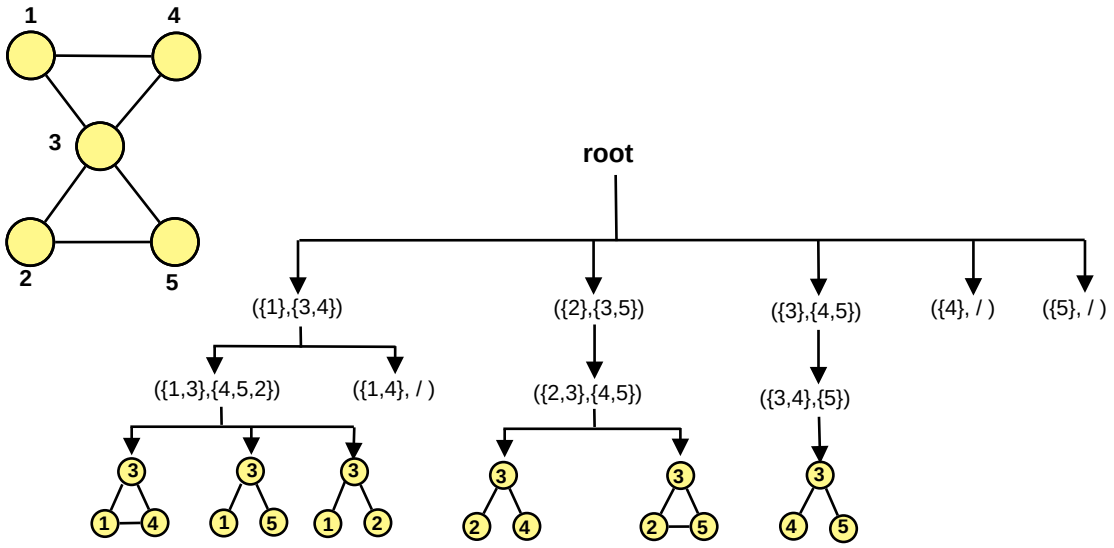

Figure 3.1: An ESU- Search Tree corresponds to calling countSubgraphs$(G, 3)$.

that later we switch from Nauty to VF2. There are many algorithms that discover isomorphism between graphs, the following sections describe one previously designed and two more recent ones.

Ullman's Algorithm This algorithm was proposed by Ullman dates back to 1976, and it forms the basis of many subgraph/graph isomorphism algorithms [Ull76]. In Ullman's Algorithm, permutation matrices are generated for the target graph $G2$ and these matrices are compared with the subgraph $G1$ to check isomorphism between $G1 and G2$. Its time complexity is $O(m^n n^2)$ and its space complexity is $O(n^2 m)$ where m,n are the orders of the subgraph and graph, respectively [Erc15].

Nauty Algorithm Nauty algorithm proposed by McCay, it is a powerful algorithm in finding graph and subgraph isomorphism [MP14]. Before testing for isomorphism Nauty algorithm transforms the graphs to be matched to a canonical form, and while searching for isomorphism it uses vertex invariants and group theory. A partition P of a graph G divides its vertices into disjoint sets of vertices $V1, ..., V_m$. A *leaf partition* is a partition that has only singleton sets. Nauty computes the invariants over the whole graph forming an initial partition, and then it computes the invariants for the individual partitions to distinguish them. The two main methods in Nauty are refining a partition and generation the children of a partition. Nauty does a depth-first-search of the space partitions, it refines each partition before expanding its children as shown in algorithm 3.2 [Erc15]. A detailed explanation of Nauty can be found in [For96].

VF2 Algorithm VF2 algorithm was introduced by Cordella et al. for exact matching

Algorithm 3.2 `Nauty`: Algorithm for finding the canonical representation of a graph $G1$.

Input: $G_1(V_1, E_1)$
Output:$G_2(V_2, E_2)$: a canonical graph

$P \longleftarrow$ partition of a single cell V
$S \longleftarrow$ stack containing P
while $S \neq \emptyset$ **do**
  if $u =$ leaf partition **then**
    UPDATE$(G2, u)$
  else
    REFINE(u)
    append children of u to S

in graphs [CFSV04]. VF2 is based on depth first search and reduces the search tree using a refinement procedure. If there are two graphs G1(V1,E1), G2(V2,E2) then the mapping M between them is defined as a set of vertex pairs(u,v) with u ∈ V1 and v ∈ V2. G1 and G2 are considered isomorphic if and only if the mapping M is a bijective function and preserves the branch structure of the graphs G1 and G2. The mapping function does not have any component in the initial state $s0$, an isomorphic is found if the partial obtained mapping covers all the vertices of G1; otherwise, a new pair of vertices is added to the current state enlarging the search tree by DFS.

This algorithm introduces a set of rules, and the candidate pair of vertices p is tested by a number of pruning rules, p is found compatible with the existing subgraph if it passes all of these tests. The time complexity of this algorithm in the worst case is $\Theta(n!n)$, and in the best case is $\Theta(n^2)$. Space complexity is $\Theta(n)$ [CFSV04]. Algorithm 3.3 shows how VF2 works to decide if two graphs are ismorphic [Erc15].

A performance comparison done by Foggia [FSV01] says that the VF2 algorithm obtains the best performance for graphs of small size and for quite sparse graphs, while for dense graphs the Nauty algorithm obtains the best results.

3.2.2 G-Tries

G-Tries is a very efficient algorithm for network motifs discovery. The idea in this algorithm is to find a set of subgraphs that is not necessary the all possible subgraphs of that size, and not only one subgraph type at a time. In comparison with previously existed algorithms, G-tries is at least two times faster [RS14b], its efficiency is because of the g-trie data structure which is used as the search space. This identifies the

Algorithm 3.3 VF2: Algorithm for deciding if two graphs $G1$ and $G2$ are isomorphic.

```
1: procedure MATCH(G1, G2, s)
2: Input: Two graphs G1 and G2 and an intermediate state s
3: Output:The mapping between the two graphs G1 and G2
4:
5:     if M_s covers all nodes of G1 then return M_s;
6:     else compute P(s) of candidate pairs to be included in M(s)
7:         for all p ∈ P(s) do
8:             if p is compatible with M_s then
9:                 s' ⟵ p ∪ M_s
10:                compute state s' obtained by adding p to M_s
11:                MATCH(G1, G2, s')
12:
           restore data structures
13:
```

common substructures between the different subgraphs that are being searched in the complex network. In the following sections we present the data structure that is used at the core of this algorithm, the g-trie. We discuss how g-tries are created, how they can be used for subgraph census and how their structure offers opportunities for parallelism.

3.2.2.1 Structure

G-Trie is a similar concept to prefix tree [Fre60], prefix trees were introduced by Fredkin in 1960, they are trees where all the descendant nodes of a node share the same common prefix. Figure 3.2 shows an example of prefix tree for storing sequences and some words. It shows also how the same prefixes are aggregated in the same node. This structure is efficient for searching, and it saves space because the common prefixes are stored only once.

Furthermore, in the case of graphs, two or more graphs can share smaller subgraph. G-trie is a prefix tree where each node of the tree represents a vertex in the graph. The connection between each vertex and its ancestor nodes characterizes it. These connections are represented as an array of boolean values where the first value of the array is one if this vertex is connected to the first node and zero if they are not connected, the second value represents the connection to the second node etc. So all graphs with common ancestor nodes share the same substructure of the tree. Figure 3.3 shows an example of a g-trie representing 6 undirected subgraphs of size 4.

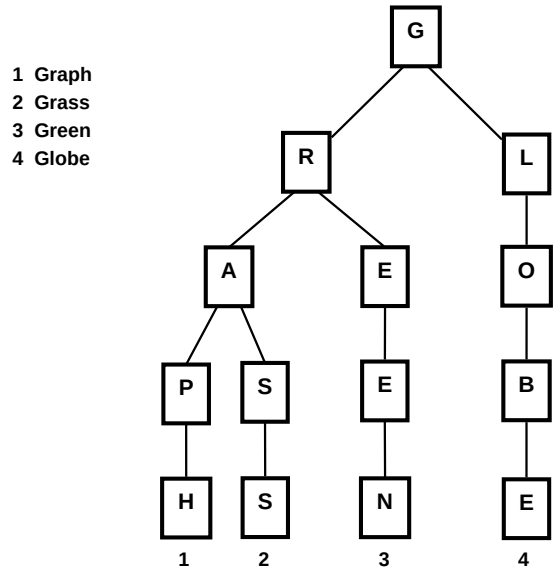

Figure 3.2: A prefix tree representing a set of four words.

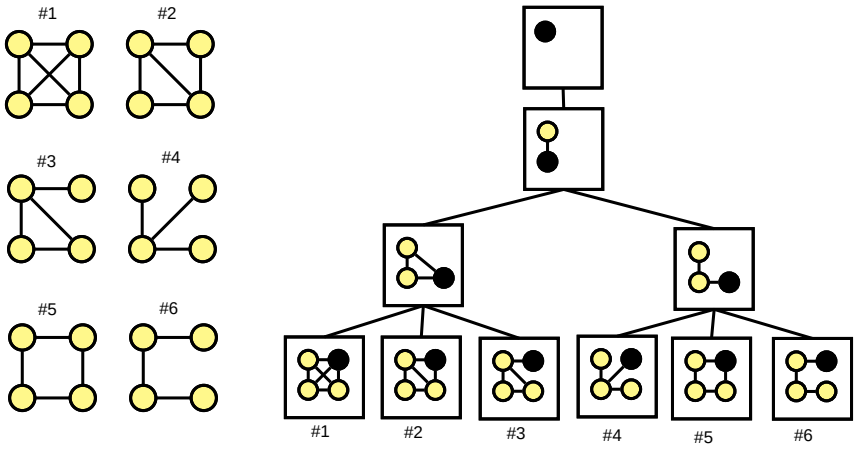

Figure 3.3: A g-trie representing a set of 6 undirected subgraphs of size 4. Taken from [RS14b]

The definition of the **g-trie** is: "A multiway tree that can store a collection of graphs. Each tree node contains information about a single graph vertex, its corresponding edges to ancestor nodes and a boolean flag indicating if that node is the last vertex of a graph. A path from the root to any g-trie node corresponds to one single distinct graph. Descendants of a g-trie node share a common subgraph" [RS14b].

3.2.2.2 Creating the G-Trie

The first step in counting occurrences using G-Tries is to build the g-trie tree that represents the possible subgraphs, building it happens by repeatedly insert one subgraph at a time. The operation starts with a tree with empty root node, during the insertion of a subgraph the tree is traversed to verify if any of the children has a the same connection to the previous nodes as the graph being inserted. The iterative insertion is defined by the adjacency matrix representing the graph being inserted. The problem is that there may be many adjacency matrices representing the same isomorphic class of graphs. However, to avoid that, g-tries uses canonical labeling to make sure that isomorphic graphs will always have the same adjacency matrix then they will generate the same g-trie representation.

The g-trie uses its own canonical representation to be more efficient, the idea is that it is an iterative algorithm where in each iteration it chooses one node and label it with the last available labeling position and deletes it from the original graph. The chosen is the one with the least number of connections. then the lookup tables are updated.

By increasing the amount of common ancestor substructures, the size of the tree is decreased and therefore the memory needed is decreased too. the following equation tells how much memory is saved also it shows the identified common substructure.

$$compressionratio = 1 - \frac{nodesInTree}{\sum nodesOfStoredGraphs}$$

The goal is to maximize the compression ratio to reduce the amount of needed memory as possible, also that will reduce the required time to count frequencies, taking the advantage that the realworld networks are usually sparse [RS14b].

In order to avoid subgraph symmetry, in an efficient way g-tries generates symmetry breaking conditions for each subgraph. It uses a procedure close to what was done by [GK07]. It generates a set of "less than" conditions between the labels of the vertices which illustrates their index, such that if a vertex a is less than a vertex b that means the index of a is smaller. The symmetry breaking conditions are stored in each node in order to use them, those conditions make the operation of counting occurrences more efficient.

Using the canonical representation and the symmetry breaking conditions, g-tries make sure that each subgraph will be found only once, and the search is faster because some branches will be cut and not explored when the condition is broken and that is discovered early in the recursion.

3.2.2.3 Computing subgraph frequencies

After having the g-trie constructed, it is ready to count occurrences of subgraphs. The retrieval of the initial set of graphs can be done easily. For instance, a path from the root to any other node at depth k with isGraph property set to true represent a graph with k nodes.

It does two operations on the same time, it backtraks through all possible subgraphs and apply the isomorphism tests while constructing the candidate subgraphs. Algorithm 3.4 explains how G-tries finds and counts all occurrences of a set of subgraphs represented in the g-trie T in a graph G. The currently partial match of the graph vertices to a path in the g-trie is reepresented by V_{used}. The algorithm starts with the children of the root of the g-trie (line: 3), and for each vertex in the graph it calls the recursive census function **COUNT** with a matched set containing only this vertex (line: 4). The **CENSUS** function starts by creating a set of vertices that completely match the current g-trie node (line: 6). Then this set is traversed (line: 7) and recursively expanded throw all possible tree paths (lines: 11,12), if the node is a leaf this means it corresponds to a full subgraph and the frequency is incremented.

Method **matchingVertices()** generates the set of candidate vertices that match the required conditions.

It starts by creating the set of candidates (ln :14), by selecting from the matched vertices which are connected to the current vertex, the vertex with the smallest number of neighbors is chosen to reduce the number of candidates to the unused neighbors. then the set of candidates is traversed (ln:16) and each vertex that respects all connections to the ancestors is added to the set of matching vertices.

Since lexicographically larger representation is used, the initial nodes will have the maximum possible number of connections and this constrains the search and reduces the possible matches.

The efficiency of G-tries made us choose to parallelize it. Globally, g-tries are 30.1 times faster, on average, than its competing algorithms.And with a global average speedup of 44.3 against the competing algorithms[RS14b].

3.3 Parallel Algorithms

We discussed in the previous chapters the importance of subgraph census and motif discovery on one hand and parallel computing advantages on the other hand. However,

Algorithm 3.4 `gtrieScanner`: Algorithm for computing the frequency of subgraphs of a g-trie T in graph G.

```
 1: procedure COUNTALL(T, G)
 2:     for all vertex v of G do
 3:         for all child c of T.root do
 4:             COUNT(c, {v})
 5: procedure COUNT(T, V_used)
 6:     V ← MATCHINGVERTICES(T, V_used)
 7:     for all vertex v of V do
 8:         if T.isLeaf then
 9:             T.frequency++
10:         else
11:             for all child c of T do
12:                 COUNT(c, V_used ∪ {v})
13: function MATCHINGVERTICES(T, V_used)
14:     V_conn ← vertices connected to the vertex being added
15:     m      ← vertex of V_conn with smallest neighborhood
16:     V_cand ← neighbors of m that respect both
17:              connections to ancestors and
18:              symmetry breaking conditions
19:     return V_cand
```

parallel algorithms in subgraph census problem are scarce.

3.3.1 Parallel approaches for ESU algorithm

Researchers tried to parallelize ESU algorithm using different approaches. Ribeiro, Silva and Lopes implemented Parallel distributed memory algorithms [RSL12], using MPI for communication. They proposed two strategies a master-worker and a distributed control, in which a randomized receiver is employed to provide a dynamic load balancing during the computation process, and they got almost linear speedups. A previous work was done by the same authors that does a single complete subgraph census [RSL10b]. Before that, other research tried to parallelize the *Grochow*algorithm in motif discovery, including both query parallelization, where different subgraphs are searched for in parallel, and network parallelization, where the original network is partitioned into overlapping regions, and a single subgraph is searched for in parallel among the different regions [SCBB08].

In 2005 a parallel algorithm that count motifs up to size 6 appeared [WTZ$^+$05], it depends on the neighborhood of the vertex that was defined by the author, their method seeks a neighborhood assignment for each node.

3.3.2 Parallel approaches for G-Tries algorithm

There are not many parallel approaches for G-Tries algorithm. In 2010 parallel distributed memory algorithm has been implemented using MPI for communication [RSL10a]. This algorithm depends on randomized receiver-initiated load balancing, it has the ability to stop the computation at any time and divide its remaining search space into two parts, give one part to the receiver asking and continue its work with the second part. It was an efficient algorithm and they obtained near-linear speedup up to 128 processors. A different parallel algorithm was done using P-threads [ARS14], it divides the work during the execution dynamically using a diagonal work sharing strategy. Almost linear speedup was obtained up to 32 cores, and a speedup over 50 for 64 cores.

3.3.3 MapReduce parallel algorithms for subgraph search

There is not any algorithm that does parallelisation for G-Tries algorithm using MapReduce. However, there are two algorithms that parallelize ESU using MapReduce. Moreover, there are some MapReduce algorithms for different subgraph search algorithms, mainly for subgraph-centric ones (search one subgraph at a time). Most of them depend on a decomposition strategy where they decompose the searched subgraph into smaller subgraphs and in each MapReduce iteration join those small instances to form bigger ones and keep iterating until reaching the required size. Following examples of those algorithms:

3.3.3.1 MRSUB

MapReduce approach was proposed by Shahrivari, S., & Jalili, S. they presented a distributed solution for subgraph discovery, they do something close to ESU but their way to satisfy the load balancing in enumerating subgraphs is by extending edges instead of vertices. [SJ15].

3.3.3.2 Iterative Hadoop MapReduce

Recently Verma, Vartika, Paul Park Kwon, and Wooyoung Kim implemented an iterative mapReduce algorithm[VKK15], they had speedups between 1.33 and 37 using 57 cores. Their procedure was to count subgraphs of size 2 then enhance the size of subgraph by one in each step and repeat until the input size of the subgraph is reached. This way results in having a large intermediate data to be processed, which expresses memory constrains. Because of that their best speedups were when the subgraph size

was small (it was 4). A small comparison between their work and ours is shown in chapter 5.

3.3.3.3 Edge-based join

Edge-based join approach was proposed by Plantenga [Pla13], this is a subgraph-centric approach in which the pattern subgraph is decomposed into an ordered list of edges, and the enumeration is done by multible MapReduce iterations, where in each iteration uses the join operation to grow one edge. This approach is not efficient, because joining one edge in each round cannot fully make use of the structural information of the pattern subgraph, which may render numerous partial results. Its decomposition strategy can be explained via the following example. For a square pattern P where $V(p) = v1, v2, v3, v4$ and $E(P) = (v1; v2); (v2; v3); (v3; v4); (v4; v1)$, the optimal pattern decomposition based on EdgeJoin is$p0 = (v1; v2); p1 = (v2; v3); p2 = (v3; v4); p3 = (v4; v1)$. However, using this pattern-decomposition strategy, the algorithm executes in three MapReduce rounds.

3.3.3.4 Star join

A starJoin approach [SMK93] is a subgraph-centric approach that follows a similar strategy to the EdgeJoin one, which depends on deviding the pattern subgraph, the difference is in the composition of the subgraph. It is considered better than EdgeJoin because it can finish in fewer MapReduce rounds, however, StarJoin still suffers from the scalability problems due to the generation of many matches when evaluating a star with many edges. For example, in a social network such as Twitter, it is very common for a node to have more than 10,000 followers. As a result, this node with its followers will contribute to over 10^12 matches of a single star p0.

3.3.3.5 TwinTwig join

TwinTwigJoin is a subgraph-centric algorithm that follows the left-deep join framework. It does the TwinTwig decomposition using an A*based algorithm, in order to decompose the searched subgraph. They perform three optimization strategies to further improve their subgraph enumeration algorithm, namely, order-aware cost reduction, workload skew reduction, and early filtering. In comparison with the metioned algorithms that do decomposition (Edge-based join and Star join), the TwinTwigJoin is faster according to their study [LQLC15].

3.3.3.6 Multiway

A multiway approach does the subgraph enumeration only in one MapReduce round [AFU13]. It duplicates each edge in different machines so that each machine can enumerate the instances independently and no match is missed. However, this approach encounters serious scalability problems since it keeps the graph in the memory of each machine. That makes it not scalable for either large data graph G or complex pattern subgraph P.

Parallel Approaches 4

4.1 MapReduce

4.1.1 Motivation

MapReduce, which was developed by Google, is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks[DG08]. Before developing it Google needed to process large amounts of raw data for instance, web requests logs and crawled documents to compute various kinds of derived data as inverted indices, summaries of the pages crawled per host and the frequent queries in a specific day. This type of computations is usually conceptually straightforward, but the input data is high. Furthermore, those computations should be done in a reasonable amount of time, then the work needs to be distributed across hundreds or thousands of machines. The issues of how to parallelize the work, distribute the data and handle the failures with large amounts of complex code were complex, thus MapReduce was developed as a reaction. MapReduce hides the messy details of parallelization, data distribution, fault tolerance and load balancing in a library. It is also a powerful interface that enables automatic parallelization and distribution of large-scale computations. Moreover it can be used to distribute the work across different clusters and also across different cores in the same machine. Finally, the MapReduce distributed programming paradigm has become increasingly supported by companies and computing clusters[DG08].

4.1.2 Programming model

One of the special points of this computation is that it takes as input a set of key/value pairs and produces as output a set of key/value pairs too. MapReduce operation is composed of two functions the Map and the Reduce, so the user of the MapReduce library has to express the computation into those two functions. Map function takes the input which is a set of key/value pairs and produces as output an intermediate set of key/value pairs, then the reduce function takes the intermediate key/value pairs,

usually each reducer takes the pairs which have the same key and its job is to merge them to reduce the set of values for that key and produce zero or one output value per reduce invocation [DG08].

4.1.3 Iterative MapReduce

Iterative MapReduce is useful for many types of problems including graph problems. The idea is to call the same Mapper and Reducer multiple times with small alteration such as the input data. In Fact, each iteration can use the output of the previous iteration as its input. A simple loop condition is used to proceed the iteration or break the loop, this condition can be set by the mapper, the reducer or the driver.

4.1.4 Frameworks

4.1.4.1 Hadoop

Hadoop is an open source Big Data framework [Apa16a], it is extremely famous and widely used. However it has some drawback, Hadoop MapReduce is difficult to program and needs abstraction, does not leverage the memory of Hadoop cluster to the maximum, users just get to process a batch of stored data and it is totally disk oriented. Hadoop Pipes can be used to run C++ code on hadoop cluster [Col16]. Since our code was originally implemented using C++, we tried it but it did not match our needs.

4.1.4.2 MR4C (MapReduce for C)

MR4C is a framework developed by Google, it is a C wrapper that allows runing native code within the Hadoop cluster, pairing the advantages of native algorithms like performance and flexibility with the advantages of Hadoop like scalability and throughput. Therefor MR4C is used in large-scaling advanced data processing algorithms [Goo15].

MR4C does not have a driver (master worker controlling the mappers and reducers), its mapper and reducer functions are called from a script, because of that the format of the key/value pairs, it accepts, is different as well. Actually, it needs an input and output directories to be identified. For this reason MR4C's mappers take as input the collection of files which exist in the input directory, then it forms the key value pairs as follows, the key is the name of the file and the value is its content. The mappers run with the mentioned input data, and output key-value pairs (files) to the intermediate directory, where the key is the name of the new file and the value is the result of the mapper function on the corresponding data.

Then the reducers use those intermediate files as input key/value pairs, apply its function and write the output as files to the output directory, the number of output files equals the number of reducers specified by the user.

In our research, we were motivated to use MR4C because the existed implementation of the G-Tries and ESU algorithms was in C++. However, that was not a good choice because MR4C is built on top of Hadoop which is disk oriented, then all the files that were mentioned need to be read and written to the disk using HDFS, and these operations are time consuming ones. Also our strategy is iterative MapReduce, as a result many MapReduce operations are going to run which highly increases the effect of writing to the disk. Moreover in our algorithm the first step is to build the graph, and that step needed to be repeated in every iteration because of the absence of a driver (master worker) in MR4C. These consequences were incurring a significant performance penalty, so MR4C did not meet our needs.

4.1.4.3 Spark

The previous systems are built around an acyclic data flow model, but there are many popular applications that require a cyclic data flow. Spark focus on those applications that reuse the same working set across multiple parallel operations and it provides similar scalability and fault tolerance like MapReduce [ZCF+10]. Unlike Hadoop, Spark is easy to program,could be used to perform streaming, batch processing and machine learning in the same cluster, also Spark batch processing is 10 to 100 times faster than Hadoop MapReduce.

Spark is the first system that allows the usage of general-purpose programming language in processing large datasets on a cluster. Spark ensures low latency computations by caching the partial results across its memory of distributed workers. Resilient distributed dataset (RDD) is the main abstraction in Spark, it represents a read only dataset that can be rebuilt if a part of it was lost, if a partition was lost RDD has enough information to rebuild only that partition. RDD can be cached in memory and used in multiple MapReduce like parallel operations.

Apache Spark by now has a huge community of users, because programming with Spark using Scala is much easier, and it is much faster than the Hadoop MapReduce framework both on disk and in-memory. The mentioned advantages of Spark made us do a Java implementation for the algorithms we are parallelizing to use it.

4.2 Parallel Approaches

The main bottleneck in subgraph search is the subgraph enumeration which consumes 95% of the execution time. Therefor the key issue in to parallelize the enumeration computation. This section will explain how we parallelize it using MapReduce. The evolution of our strategy passed two main approaches, that will be described in the following sections and for each approach a detailed explanation is given about how we applied it on our case studies ESU and G-Tries.

4.2.1 Static division approach

This is the naive approach, the idea here is to statically divide the vertices between the mappers such that each mapper processes its portion and returns the result of that portion, then the reducers build the whole final result. This is possible because in tree-like algorithms the enumeration of subgraphs of each vertex is independent, so calculating the subgraphs starting with any vertex does not depend on any operations on different vertices. So we divide the vertices between the mappers and then each mapper enumerates the subgraphs that are under that vertex in the graph, then the reducers sum those results to display the final result. We decided to divide the nodes in a round robin way because normally the amount of work in the root nodes is sharply higher than in the leaves, that is because of the breaking similarity conditions for instance in ESU tree to expand a vertex it should has a higher label than the label of the vertex being processed.

The drawback in this way is that the amount of work is unbalanced between workers (mappers) since graphs are generally unbalanced and the number of subgraphs rooted from each vertex differs a lot and it may be the case that more than 20% of the work is under one node.

The following sections describes how we applied this approach on our case studies.

Applying it on ESU

By considering the approach described in algorithm 4.1, the method **main** is the driver, it first divides the nodes in a round robin way into a set of work-sets(lines: 8-9) then each work set will be assigned to a worker. The driver initiates mappers by calling the map function (line: 10). Each mapper executes the method **countSubgraphs** with a set of vertices ($vSet$) to calculate the number of occurrences under them. The driver collects the results from the mappers using the reducer (line: 11) which combines the

partial trees into one tree that has all the occurrences, which are **not isomorphic** yet.

Subgraph Isomorphism tests consume a lot of time because ESU algorithm originally applies isomorphism tests every time it finds a subgraph as explained in section 3.2.1. Toward reducing that time we apply the idea used in FaSE algorithm [PR13] as follows, after the reducer finishes we have only one result tree containing all the occurrences.

Then this tree is populated to extract the subgraphs stored in it (line: 12), and a new MapReduce step starts to do the isomorphism tests between these subgraphs (lines: 16,17). Each mapper now will take the same set of subgraphs and compare each of them with the rest using VF2 algorithm to detect isomorphism. When two subgraphs are isomorphic the mapper sum their occurrences and save the id of the second one to be deleted by the reducer in the driver. After we do all the required isomorphism test, we print the last final result (line: 19).

Using the tree reduces the number of isomorphism tests because the number of leaves in the tree is less than the number occurrences. An example that shows the difference between the number of leaves and the number of occurrences for different subgraph sizes in two different networks is shown in the table 4.1.

Network	Subgraph size	Subgraph types	Number Of Occurrences	Number of Leaves
Power	3	2	17,631	3
	4	6	63,401	17
	5	21	268,694	170
	6	101	1,260,958	1,771
	7	626	6,340,413	14,441
	8	4,516	33,494,650	96,219
	9	31,543	183,453,978	565,387
Foldoc	3	13	2,553,830	45
	4	198	228,272,189	2,304
	5	8,345	29,621,881,964	141,115

Table 4.1: Examples for the difference between the number of occurrences and number of leaves in the corresponding tree. Adapted from [PR13].

Algorithm 4.1 ESU: Static-division mapReduce parallel algorithm for computing the frequency of subgraphs of size k in graph G using w workers.

1: **procedure** MAIN(k, G, w)
2: $result \longleftarrow$ A tree that will contain the total result.
3: $verticesSets \longleftarrow$ a list that will contain the work-sets for workers, its length is w.
4: $partialResult \longleftarrow$ The tree that will contain the result of each worker.
5: $nIsoOccs \longleftarrow$ List of not isomorphic occurrences
6: $finalOccs \longleftarrow$ List of isomorphic occurrences
7: $ctr = 0$
8: **while** $ctr < G.numVerices$ **do**
9: $verticesSets[ctr \mod w].add(ctr)$; $ctr++$
10: $partialResult =$ VERTICESSETS.MAP(vSet)COUNTSUBGRAPHS($G, k, vSet$)
11: $result = partialResult$.REDUCE($partialResult1, partialResult2$)
12: nIsoOccs = POPULATEMAP($result$)
13: $ctr = 0$
14: **while** $ctr < nIsoOccs.length$ **do**
15: $isoSet \longleftarrow$ set of ni isomorphic subgraphs from $nIsoOccs$
16: $partialOccs =$ WORKERS.MAP($isoSet$)APPLYISO($isoSet, nIsoOccs$)
17: $finalOccs \mathrel{+}= partialOccs$.REDUCE($partialOccs1, partialOccs2$)
18: $ctr+ = ni$
19: **Print** $finalOccs$
20: **procedure** COUNTSUBGRAPHS($G, k, vSet$) $partialResult$=null;
21: **for all** vertex v of $vSet$ **do** $EXT \leftarrow u \in N(V) : v < u$
22: $partialResult =$EXTEND(v, $\{v\}, EXT, partialResult$)
return $partialResult$
23: **procedure** EXTEND($v, V_{subgraph}, EXT, partialResult$)
24: **if** $|V_{subgraph}| = k$ **then** partialResult $+= V_{subgraph}$;
25: **for all** vertex v in EXT **do** $EXT2 = EXT \cup \{u \in N_{excl}(w, V_{subgraph}) : u > v\}$
26: partialResult += EXTEND($V_{subgraph} \cup \{w\}, EXT2, v, partialResult$)
return partialResult
27: **procedure** APPLYISO($isoSet, nIsoOccs$)
28: $toDelete \longleftarrow$ Holds the subgraphs' ids that should not be compared any more.
29: $mid = mapper.getID$
30: $ctr = mid$
31: **for all** subgraph $sub1$ in $isoSet$ **do**
32: **while** $ctr < nIsoOccs.length$ **do**
33: $sub2 = nIsoOccs.get(ctr)$
34: **if** VF2.AREISOMORPHIC($sub1, sub2$) **then**
35: $isoSet.update(sub1, sub1.occurrences + sub2.occurrences)$
36: $toDelete+ = sub2.id$
37: $ctr+ = mid$
return $(isoSet + toDelete)$

Applying it on G-Tries

Algorithm 4.2 `G-Tries`: Static-division, parallel MapReduce algorithm for computing the frequency of subgraphs of a g-trie T in graph G using w workers.

```
1:  procedure MAIN(G, T, w)
2:      result ⟵ A g-trie that will contain the total result.
3:      verticesSets ⟵ a list that will contain the work-sets for workers, its length is w.
4:      partialResult ⟵ The g-trie that will contain the result of each worker.
5:      ctr = 0
6:      while ctr < G.numVerices do
7:          verticesSets[ctr mod w].add(ctr) ; ctr++;
8:      partialResult = verticesSets.map(vSet)countAllMapper(T, vSet)
9:      result = partialResult.reduce(partialResult1, partialResult2)
10:     Print result
11: procedure countAllMapper(T, vSet)
12:     resArr ⟵ the auxiliary array.
13:     for all vertex v of vSet do
14:         for all child c of T.root do
15:             count(c, {v}, resArr)
        return resArr
16: procedure count(gN, V_used, resArr)
17:     V ← matchingVertices(gN, V_used)
18:     for all vertex v of V do
19:         if gN.isLeaf then
20:             resArr[gN.id]++
21:         else
22:             for all child c of gN do
23:                 count(c, V_used ∪ {v}, resArr)
24: function matchingVertices(T, V_used)
25:     V_conn ← vertices connected to the vertex being added
26:     m      ← vertex of V_conn with smallest neighborhood
27:     V_cand ← neighbors of m that respect both
28:              connections to ancestors and
29:              symmetry breaking conditions
30:     return V_cand
```

As we can see in algorithm 4.2 the driver builds firstly the graph and the g-trie. Then it divides the graph vertices in a round robin way between a number of work sets equal the number of workers (lines: 6-7). After that, each worker will execute the

countAllMapper to calculate the occurrences under its work set. In the end, the reducers collect the intermediate results and build a complete tree with the complete final result (line: 9). Isomorphism is insured because of the trie data structure and its conditions as explained in *chapter 3*. An important thing to mention is that, while constructing the g-trie we are assigning a key to each node, keys start from zero. And when the workers are lunched, each worker creates an array with number of entries equal the size of the g-trie, to store the occurrences that this worker finds. Every time the worker finds an occurrence, it increases the corresponding index to the g-trie's leaf node (line:20). Then the workers return those arrays to the reducer which merge them forming the final result.

4.2.2 Iterative with time limit (threshold) approach

As in the previous approach, the nodes are divided between workers in a round robin way and the workers start with an equal amount of nodes to process. After a fixed amount of time all of the workers stop and save and return their current state to the reducer, which collects the saved states. Then the remaining work is divided between workers and a new MapReduce iteration starts. This approach fixes the weaknesses of the first approach which resulted from unbalanced work, reducing the idle time (time when some workers are idle).

This threshold divides the iterative approach into two sub-approaches which are either a **fixed** threshold, or a **dynamic (or adjustable)** threshold. The problem with **fixed** threshold is that in case it is low, the number of iterations will be high, that means the idle time will be very small but a lot of time will be spent to save the state and redivide work. On the other hand, if the threshold is high, as long as the tree is unbalanced, many workers will go idle. This strategy was not giving us good speedups, because of that we developed our last algorithm that adjusts the threshold before every iteration.

In this approach we tried to decrease the weaknesses previously mentioned by making the threshold dynamically changing. It starts with an initial value equals to "$(numNodes^2) * (motif_size^3)$", then after each iteration we calculate the number of workers who finished before and the amount of time they were idle, then according to this information we reset the threshold (either increase or decrease). The threshold is readjusted as follows; If non of the workers finished before, that means the threshold is low, so we increase it by 20%. Otherwise we decrease it using the following formula, $newThreshold = ((oldThreshold * numWorkersWhoWaited)/totalNumWorkers) - (0.2 * averageWaittedTime)$. We explain why we chose those values later in *chapter 5.*

Applying it on ESU

Algorithm 4.3 details our strategy. First, as in the static division, the driver constructs the initial work sets (lines: 8-9). It sets the initial time limit (line: 7). Procedure **extend** is able to stop when the time limit exceeds and store the recursion(line:32-34). In this case, the driver builds the corresponding set of work units and stop the search. New line was added to the **countSubgraphs** procedure to test time limit (line:26), if it was exceeded, it stop doing recursive calls, break the cycle, and return the partial result combined with the saved state and remaining vertices. The driver calls the workers and after each iteration it checks if there is more work to do, if yes it adjusts the threshold (line: 14) and keep iterating until there is no more work left. Saving the state (line: 33) is an important issue that will be explained in a following section.

Applying it on G-Tries

Algorithm 4.4 shows how we apply our starategy on G-Tries. The driver starts by setting the initial time limit (line: 5) and initial work sets (lines: 6-7). Then it lunches the workers, who will start working. If the time limit exceeds before the worker finishes its work, the worker saves its location and returns its partial result (line: 26-29). Each partial result will have two components, the found occurrences ($resArr$) and the remaining work in the corresponding work set. The Reducer collects those partial results, separates the components, adds the occurrences to the final result and constructs the total remaining work. If we still have remaining work to do, the driver adjusts the threshold (line: 12) and launches the workers with the new work sets(lines: 8-12).

Algorithm 4.3 ESU: Iterative MapReduce parallel algorithm for computing the frequency of subgraphs of size k in graph G using w workers.

```
1: procedure MAIN(w, k, G)
2:     result ⟵ A tree that will contain the total result.
3:     verticesSets ⟵ a list that will contain the work-sets for workers, its length is w.
4:     partialResult ⟵ The tree that will contain the result of each worker.
5:     nIsoOccs ⟵ List of not isomorphic occurrences
6:     finalOccs ⟵ List of isomorphic occurrences
7:     timeLimit = k^2 * (G.numVerices)^2;  ctr = 0;
8:     while ctr < G.numVerices do
9:         verticesSets[ctr mod w].add(ctr);  ctr++
10:    while verticesSets ≠ ∅ do
11:        partialResult = VERTICESSETS.MAP(vSet)COUNTSUBGRAPHS(G, k, vSet, timeLimit)
12:        result += partialResult.REDUCE(partialResult1, partialResult2)
13:        verticesSets = remainingWork
14:        timeLimit = adjustThreshold(numIdleMappers, avgWaitedTime, timeLimit)
15:    nIsoOccs = POPULATEMAP(result);  ctr = 0;
16:    while ctr < nIsoOccs.length do
17:        isoSet ⟵ set of ni isomorphic subgraphs from nIsoOccs
18:        partialOccs = WORKERS.MAP(isoSet)APPLYISO(isoSet, nIsoOccs)
19:        finalOccs += partialOccs.REDUCE(partialOccs1, partialOccs2)
20:        ctr+ = ni
21:    Print finalOccs
22: procedure COUNTSUBGRAPHS(G, k, vSet, timeLimit) partialResult=null;
23:    for all vertex v of vSet do
24:        EXT ← u ∈ N(V) : v < u
25:        partialResult =EXTEND(v, {v}, EXT, partialResult, timeLimit)
26:        if timeLimit exceeded then partialResult.add(unexploredVertices)
27:            return partialResult
       return partialCount
28: procedure EXTEND(v, V_subgraph, EXT, partialResult, timeLimit)
29:    if |V_subgraph| = k then
30:        partialResult.add(G[V_subgraph])
31:    for all vertex v in EXT do
         EXT2 = EXT ∪ {u ∈ N_excl(w, V_subgraph) : u > v}
32:        partialResult = EXTEND(V_subgraph ∪ {w}, EXT2, v, partialResult, timeLimit )
33:        if timeLimit exceeded then
34:            partialResult.add(SAVESTATE)
35:            return partialResult;
36:    return partialResult
```

Algorithm 4.4 `G-Ttries`: Iterative MapReduce parallel algorithm for computing the frequency of subgraphs of a g-trie T in graph G using w workers.

```
1: procedure MAIN(T, G, w)
2:     result ⟵ A g-trie that will contain the total result.
3:     verticesSets ⟵ a list that will contain the work-sets for workers, its length is w.
4:     partialResult ⟵ The g-trie that will contain the result of each worker.
5:     timeLimit = k^2 * (G.numVerices)^2;  ctr = 0
6:     while ctr < G.numVerices do
7:         verticesSets[ctr  mod w].add(ctr);  ctr++
8:     while verticesSets ≠ ∅ do
9:         partialResult = VERTICESSETS.MAP(vSet)COUNTALLMAPPER(T, vSet, timeLimit)
10:        result = partialResult.REDUCE(partialResult1, partialResult2)
11:        verticesSets = remainingWork
12:        timeLimit = adjustThreshold(numIdleMappers, avgWaitedTime, timeLimit)
13:    Print result
14: procedure COUNTALLMAPPER(T, G)
15:    currentState = null
16:    resArr = null ⟵ the auxiliary array.
17:    for all vertex v of G do
18:        for all child c of T.root do
19:            currentState = COUNT(c, {v}, currentState, resArr)
20:            if Time Limit exceeded then
21:                return currentState
22:    return currentState + resArr
23: procedure COUNT(gN, V_used, currentState, resArr)
24:    V ← MATCHINGVERTICES(gN, V_used)
25:    for all vertex v of V do
26:        if Time Limit exceeded then
27:            remainingV = the unexplored vertices in V
28:            currentState = SAVESTATE(remainingV, hLable, currentMatch, trieID, currentState)
29:            return currentState
30:        if gN.isLeaf then
31:            resArr[gN.id]++
32:        else
33:            for all child c of gN do
34:                COUNT(c, V_used ∪ {v}, currentState, resArr)
35: function SAVESTATE(remainingV, hLable, currentMatch, trieID, currentState)
36:    state = trieID + hLable + currentMatch + remainingV
37:    currentState += state
38:    return currentState
```

4.2.2.1 Work sets and work units

We define two keywords regarding the work assigned to each worker, work unit and work set. Work unit is a single state in the search tree from where the worker could start computing. Work set is a set of work units that is assigned to a worker. For instance, when we start our work with w workers for example, we divide the nodes in a round robin way between w work sets and each node represents a work unit. Then each worker is lunched with a work set to process. After the mappers finish their work, in case the work is not done yet, the saved states are collected and divided again into w working sets in a round robin way. Thus the next iteration begins by assigning the mappers (workers) the new working sets.

4.2.2.2 State saving

When the time threshold is reached, if the mapper has not finished his work set yet, we should stop the computation and save the search state. The goal is to save the state of the recursive work by capturing the stack contents in an efficient way. We divide the current state into a number of work sets, which is equal the number of mappers, if there is enough work. As a matter of fact, we make sure that if all the workers have finished their work except this one then in the next iteration all of them will have work to do (if there is still enough work).

ESU State

In **ESU** algorithm, in order to safe the state at the vertex where we are, we need to store the currently explored graph vertices at that level (current) and the unexplored vertices (EXTension), which is the list of the vertices that were still not traversed. Figure 4.1 exemplifies our state saving scheme, dashed nodes are the nodes that yet need to be explored. As shown in the diagram the unexplored nodes in each level are divided in a round robin way between the workers , also for efficiency reasons we save only on copy of the current per level per worker so that each work unit has compact enough information to restart computing. For example in figure 4.1 in the last level the currently explored vertices are $1, 4, 7$, so one work unit that will be received by worker one is $1, 4, 7, 11, 15$, another work unit for worker two will be $1, 4, 7, 12$. Supposing we have two workers, figure 4.2 depicts the content of the two work units in depth i.

Algorithm 4.5 simplifies the part of code in the iterative algorithm that saves the work units in a certain level of the ESU search tree. The *separator* is used to separate states from different levels (depths).

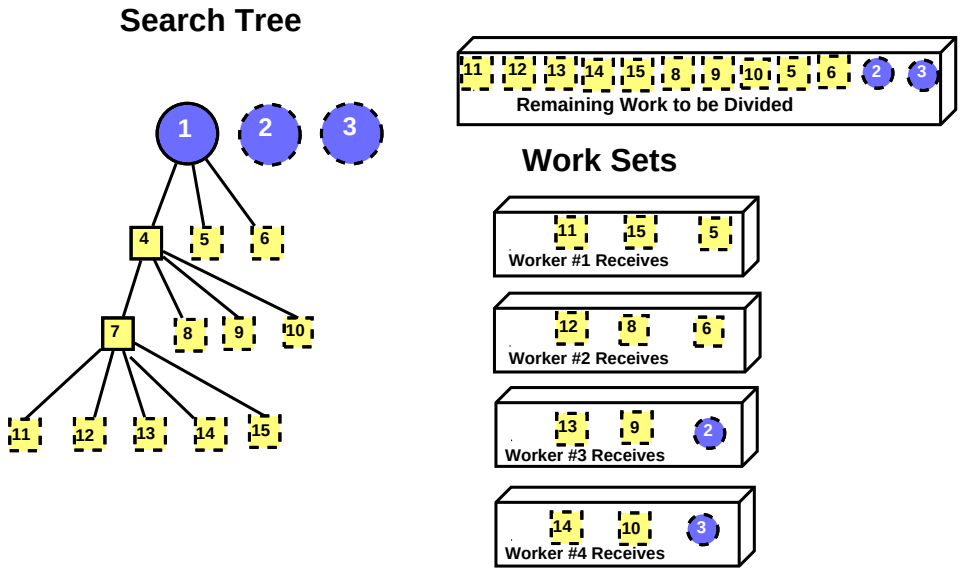

Figure 4.1: ESU search-tree and dividing the remaining work between four workers

Algorithm 4.5 `ESU`: Saving the state in ESU algorithm in a certain level.

1: $Current \longleftarrow$ the currently explored vertices.
2: $EXT \longleftarrow$ A list of unexplored vertices in this level.
3: $States \longleftarrow$ A list with a length equal the number of workers, will be filled with work units.
4: **if** $Timelimitexceeded$ **then**
5: $ctr = 0$
6: **while** $ctr < EXT.length$ **do**
7: **if** $ctr < States.length$ **then**
8: $States[ctr] += "separator" + Current$
9: $States[ctr \mod numWorkers] += EXT[ctr]$
10: $ctr++$

Vi: list of currently explored vertices, from depth 0 to depth i

UVi_k : Kth Unexplored vertex in depth i

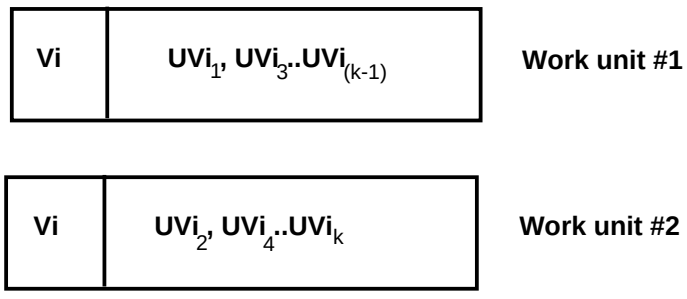

Figure 4.2: The content of the two work units in depth i

The size of the saved state will be at maximum $O(subgraphsize * V(G))$ since we can only go as far as the subgraph size and the unexplored vertices maximum will equal to the number of vertices in the graph. However, in practice the content of the state will be much lower because of the constraints applied on the unexplored vertices' list.

G-Tries State

Figure 4.3 shows the state of the recursive G-Tries search at a given time.

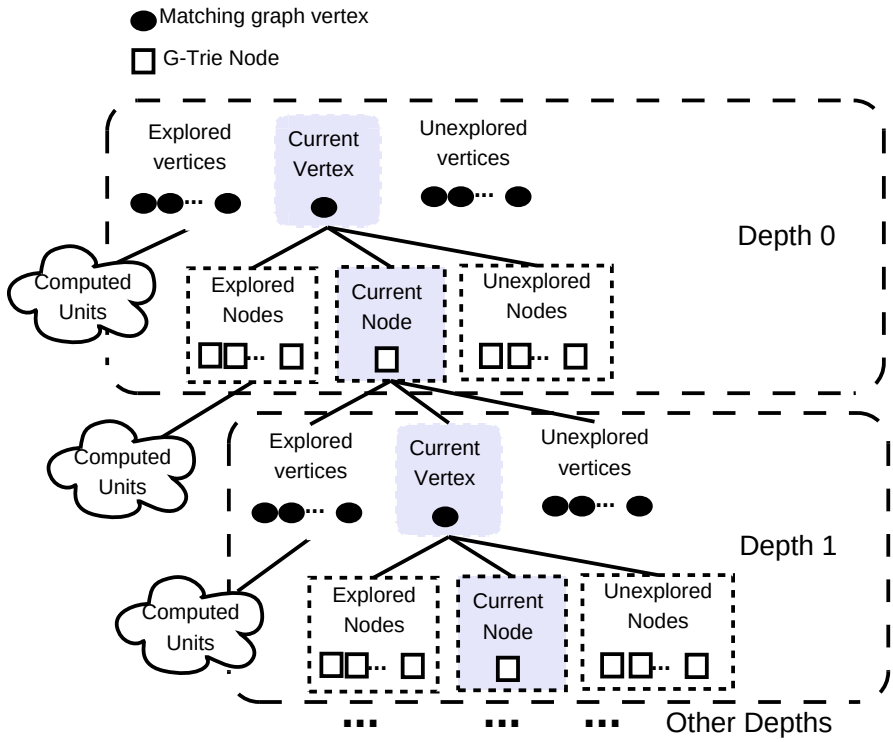

Figure 4.3: G-Tries recursive procedure frozen at a given time

The function that enumerates the occurrences in G-Tries algorithm (function **COUNT** in algorithm 4.3)contains two cycles, one enumerates all possible graph vertices and the entire loop enumerates all matching children of the corresponding g-trie node. So when we save the state in each depth we should save the position in the both cycles. Each vertex in the graph has a label to define it, and to define the g-trie node we give each one an id while constructing the g-trie before the enumeration phase. And since the g-trie is fixed, by knowing the id of a certain node we know its children. As mentioned we divide the current state at each depth into a number of working sets and each set is constructed from a set of working units where each unit has enough information to resume the work later. Supposing we have two workers, figure 4.4 depicts the content

of the two work units in depth i. Algorithm 4.6 describes the procedure applied inside the inner cycle when we need to save the state. The *separator* separates states from different depths.

Algorithm 4.6 `G-Tries`: Saving the state in G-Tries algorithm in a certain level.

```
1: Map ⟵ the currently explored vertices.
2: Fastnei ⟵ A list of unexplored vertices in this level(neighbors of the current vertex).
3: States ⟵ A list with a length equal the number of workers, will be filled with work
   units.
4: if Timelimitexceeded then
5:     ctr = 0
6:     while ctr < Fastnei.length do
7:         if ctr < States.length then
8:             States[ctr] += "separator" + Map
9:         States[ctr mod numWorkers] += Fastnei[ctr]
10:        ctr++
       return
```

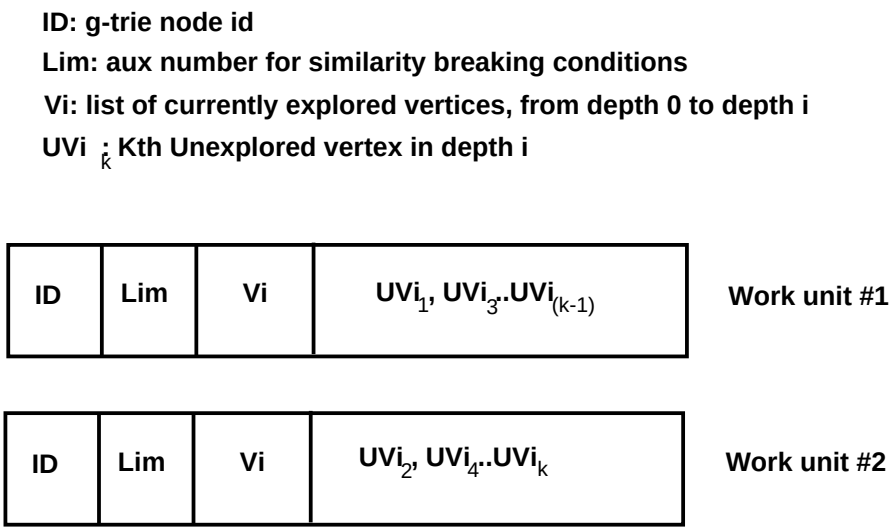

Figure 4.4: The content of the two work units in depth i

4.2.2.3 Resuming the work

As we saw in the previous section, the saved working unit has enough information to continue its work independently.

In **ESU** algorithm, the function **extend** which does the counting, takes as parameters this information. Algorithm 4.7 shows the **countSubgraphs** function that receives the working units ($vSet$) and calls the counting function. The difference between calling **extend** function for a vertex (lines: 3-5) and for a work unit (lines: (6-9) could

be seen in this algorithm too. The *current* variable expresses the currently explored vertices and it will have the ones stored in the work unit (line: 8), and EXT variable will have the vertices that need to be explored which are stored in the work unit too (line:9).

Algorithm 4.7 `ESU`: resuming the work from a saved state.

```
1:  procedure countSubgraphs(G, k, vSet, timeLimit) partialResult=null;
2:      for all item v of vSet do
3:          if v is a vertex then
4:              EXT ← u ∈ N(V) : v < u
5:              partialResult =extend(v, {v}, EXT, partialResult, timeLimit)
6:          else
7:              EXT ← v.remainingVertices
8:              current ← v.currentlyExploredVertices
9:              partialResult =extend(v, {current}, EXT, partialResult, timeLimit)
10:         if timeLimit exceeded then
11:             partialResult.add(unexploredVertices) return partialResult
        return partialCount
```

In **G-Tries**, we needed a procedure to do the bridge between the saved working unit and the recursive counting function (**count**). This procedure is shown in Algorithm 4.8. It iterates through the remaining vertices stored in the saved work unit and continue counting the occurrences by calling the original census function (lines: 10-11).

Algorithm 4.8 `G-Tries`: resuming the work from a saved state.

```
1:  procedure resumeWork(T, V_used, remainingVertices, currentState)
2:      for all vertex v of remainingVertices do
3:          if Time Limit exceeded then
4:              remainingV = the unexplored vertices in remainingVertices
5:              currentState = saveState(remainingV, hLable, currentMatch, trieID, currentState)
6:              return currentState
7:          if T.isLeaf then
8:              T.frequency++
9:          else
10:             for all child c of T do
11:                 count(c, V_used ∪ {v}, currentState)
```

4.2.2.4 Flow Diagram

Figure 4.5 illustrates our strategy.

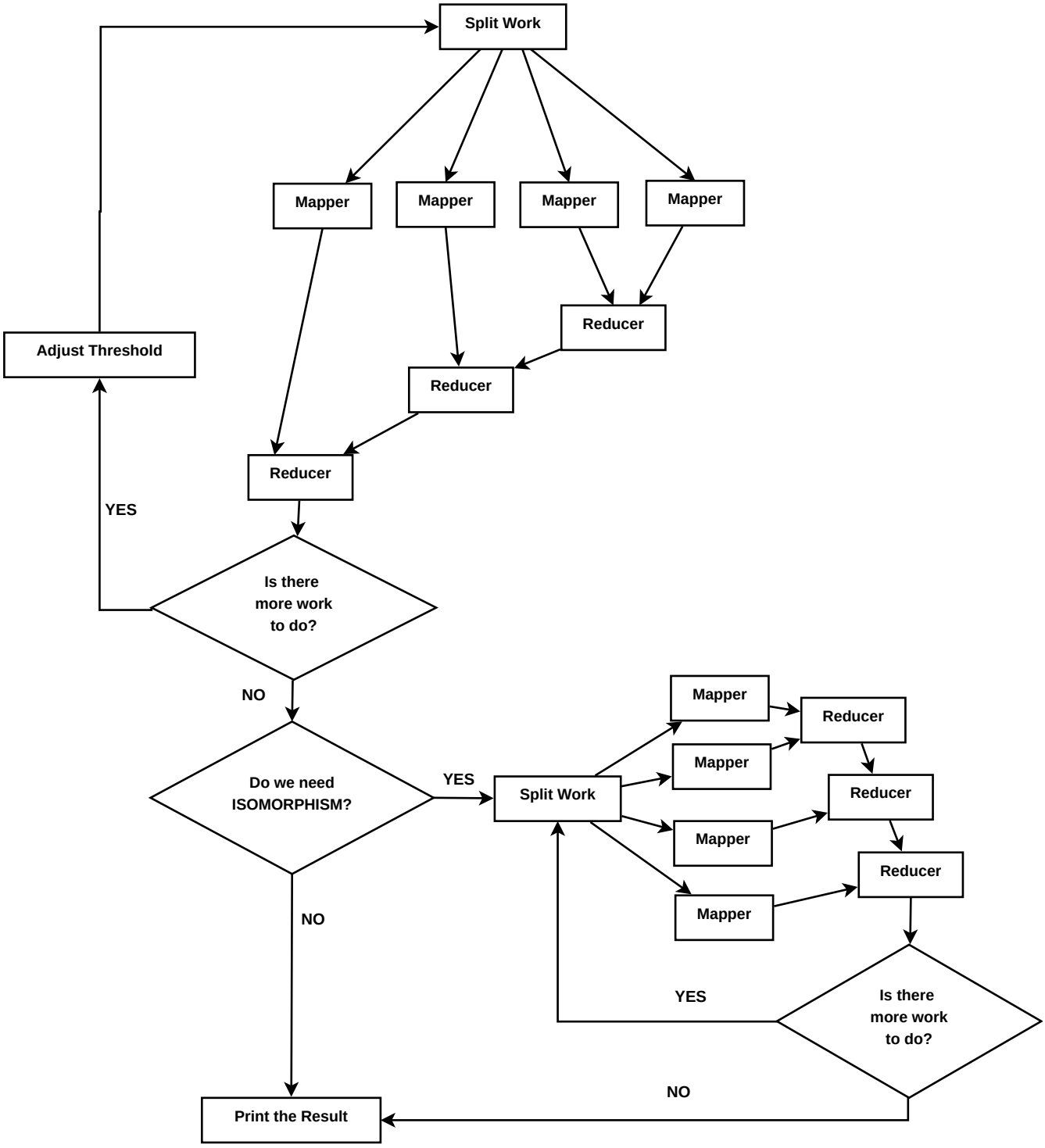

Figure 4.5: Flow chart for our strategy

Experimental Evaluation 5

In this chapter we present empirical data obtained by running our parallel methods on a large, diverse and representative sets of complex networks. First we describe the computational environment and the networks used. Then for the purpose of studying the efficiency of our algorithms, we discuss the relative overhead by comparing the original sequenial algorithms (C++) with our sequential implementation for them (Java), furthermore, we compare the sequential algorithms (Java implementation) with our parallel implementation using only one core. Finally, we do a performance evaluation for the two parallel approaches (detailed in *chapter 4*) by doing the scalability tests, showing the speedups we obtained.

5.1 Common Materials

Our experimental evaluation is organized in two major parts, each part is dedicated for one parallel approach (Static and Iterative). In each part we gathered results for the two case study algorithms (ESU and G-Tries). The two approaches were tested using the same computational environment and networks datasets which are described next.

5.1.1 Computational Environment

We obtained our results using a 64-core machine, consisting of four 16-core AMD Opteron 6376 processors at 2.3GHz with a total of 252GB of memory installed. Each 16-core processor is split in two banks of eight cores, each with its own 6MB L3 cache. Each bank is then split into sets of two cores sharing a 2MB L2 and a 64KB L1 instruction cache. A 16KB L1 data cache is dedicated to each core.

All code was developed in java and compiled using Maven3.3.9 [Apa16b], inside Spark framework. Moreover, the used time unit is the second.

5.1.2 Complex Networks

Subgraph search algorithms are ubiquitous and can be applied to any system that can be modeled as a graph, therefore the possible applications are from many different scientific fields. For the purpose of testing our work, we chose a diverse set of networks form various fields of applications, we group those networks according to their field and describe each of them. Moreover, table 5.1 shows their characteristics.

- **Social Networks**: describe relations between users from social networks. These networks are becoming increasingly popular and the study of their structure may give important insights into social organization [TMP12].
  - `facebook`: undirected network consisting of friend circles gathered from Facebook [ML12]. Source: [LK14].
- **Communication Networks**: represent networks related to communications.
  - `polblogs`: directed network of hyperlinks between weblogs on United States politics, recorded in 2005 by Adamic and Glance [AG05]. Source: [New10].
  - `company`: directed network of ownership of media and telecommunication companies [NLGC02]. Source: [BM06].
- **Biological Networks**: networks that model biological processes and concepts, their structure has been found to give important information and because of that biological networks are gaining increasing attention.
  - `neural`: directed, weighted network representing the neural network of *C. Elegans* [WS98, WSTB86]. Source: [New10].
  - `metabolic`: directed metabolic network of the small nematode round worm *C. elegans* [DA05]. Source: [Are14].
- **Semantic Networks**: represent networks related to connections between concepts.
  - `foldoc`: is a free online dictionary of computing terms, including acronyms, jargon, programming languages, tools, architecture, operating systems, networking, theory, conventions, standards, mathematics, telecommunications, electronics, institutions, companies, projects, products, history [How10]. It

is a directed network where an edge (X,Y) from term X to term Y exists in the network if the term Y is used to describe the meaning of term X. Source [BM06].

  - `september11`: is an undirected temporal network, it is based on all stories released during 66 consecutive days by the news agency Reuters concerning the September 11 attack on the U.S., beginning at 9:00 AM EST 9/11/2001. The vertices of a network are words (terms); there is an edge between two words if they appear in the same text unit (sentence). Source: [BM06].

- **Internet Networks**: represent relationships between computers, where nodes represent computers and edges communication.

  - `gnutella`: A sequence of snapshots of the Gnutella peer-to-peer file sharing network from August 2002. There are total of 9 snapshots of Gnutella network collected in August 2002. Nodes represent hosts in the Gnutella network topology and edges represent connections between the Gnutella hosts. Source: [LK14].

- **Wikipedia Networks**: Wikipedia is a free encyclopedia written collaboratively by volunteers around the world. The following networks represent relationships between Wikipedia users, where nodes represent users and edges votes.

  - `wikivote`: is a directed network that contains all the Wikipedia voting data from the inception of Wikipedia till January 2008. Nodes in the network represent wikipedia users and a directed edge from node i to node j represents that user i voted on user j. Source: [LK14].

- **Collaboration Networks**: networks consisting of relations between entities collaborating in the same subject.

  - `netscience`: undirected network containing co-authorships of scientists working on network experiments and analysis [New06]. Source: [New10].

Network	Group	$\|V(G)\|$	$\|E(G)\|$	Average Degree
neural	biological	297	2,345	7.9
metabolic	biological	453	2,025	4.7
netscience	collaboration	1,589	2,742	1.7
polblogs	communication	1,491	19,022	12.8
company	communication	8,497	6,724	0.8
gnutella	internet	8,717	31,525	3.6
foldoc	semantic	13,356	120,238	9.0
septemper11	semantic	13,314	243,447	18.3
facebook	social	4,039	88,234	21.9
wiki-vote	wikipedia	7,115	103,689	14.6

Table 5.1: The set of representative real networks used for performance evaluation.

5.1.3 Test Data

Our algorithm was evaluated up to 64 cores using the mentioned machine and networks. We searched in each network from table 5.1 for all possible subgraphs of a given size k. In Table 5.2 we show the size k used and the resulting number of all possible subgraphs of that size and type (directed or undirected) that was counted in that network. Furthermore, we show the computing time spent using our last and final approach using one core, also the growth rate is calculated and shown in order to expect the required computation for larger subgraph sizes.

Network	Subgraph Size	Number of Occurrences	Computation Time (S)	Growth Rate	
				AVG	STD
septemper11	4	9,969,545,115	2,725	305.6	3.9
foldoc	5	29,621,881,964	6,346	83.0	40.3
polblogs	5	7,347,672,714	2,662	57.5	36.1
company	5	4,335,107,042	782	59.4	74.0
gnutella	6	7,852,428,858	1,964	16.5	12.1
facebook	5	27,925,079,209	4,899	88.8	74.7
social	6	1,181,599,470	632	7.5	5.7
wikivote	4	2,513,413,248	737	56,055.7	96,449.4
netscience	9	886,423,840	2,851	452.3	1062.5
neural	7	37,818,052,163	13,215	2,690.7	5884.1
metabolic	6	9,153,235,252	2,183	552.2	1087.3

Table 5.2: `Test data`: general information about most of the tests that we did.

5.2 Overhead

In the first place we wanted our Java implementation for the sequential algorithms to be as efficient as possible in comparison with the sequential C++ implementation. However, Java is slower than C++ in general, because it was designed to be a simple language. Therefore many of the features available in C++ that give the programmer control over details were intentionally stripped away. As a result our Java implementation for G-Tries consumes double the amount of time taken by the C++ one, the overhead is shown in table 5.3. However, there is still a space to optimize our code more.

Network	Subgraph Size	Sequential C++ Time (s)	Sequential Java Time (s)	Overhead
`polblogs`	5	511	1,050	$\approx 105\%$
`company`	5	157	336	$\approx 114\%$
`netscience`	7	7	14	$\approx 100\%$
`polblogs`	4	6	12	$\approx 100\%$
`foldoc`	5	1838	3,952	$\approx 115\%$

Table 5.3: `G-Tries`: Comparison between the original C++ sequential version and our Java sequential version.

On the other hand, our Java implementation for ESU algorithm was faster than the C++ one, because of the delay of isomorophism tests as explained in `Chapter 4`. Also because we are building the subgraph incrementally on fly, so that when we reach the required depth, the subgraph will be already built and only needs to be inserted in the trie. However, when the subgraph size is large our implementation becomes slower, and that happens because we are using a different isomorphism algorithm (VF2), which is slower than the one used in the original algorithm (Nauty) when the searched subgraph size is large. For instance, in the forth example shown in table 5.4, the 539 seconds spent as follows, 18 seconds in counting the subgrahs and building the trie, and 521 seconds in doing the isomorphism tests. This problem could be solved by having a java implementation for Nauty algorithm and using it instead when searching large subgraphs. Table 5.4 shows a comparison between the both the original C++ implementation and our java implementation for ESU.

In addition, we compared the execution time of our parallel algorithms using only one core with the original sequential algorithm, in order to study the imposed overhead by our parallel solution and by Sark. Table 5.5 shows the overhead imposed by applying our second and final approach (Iterative with dynamic threshold) on G-Tries algorithm which is in average 112%, this overhead is mainly caused by the time limit tests, which

Network	Subgraph Size	Sequential C++ Time (s)	Sequential Java Time (s)	# Times Faster
foldoc	4	728	108	≈ 6.7
polblogs	4	148	26	≈ 5.7
company	4	97	12	≈ 8.1
netScience	7	87	539	≈ 0.2

Table 5.4: ESU: Comparison between the original C++ sequential version and our Java sequential version.

means the amount of time spent by each worker to compare the amount of time it has been running with the threshold.

Network	Subgraph Size	Sequential Time (s)	Single Worker Time (s)	Overhead
gnutella	6	942	1,964	≈ 108%
foldoc	5	3,952	6,346	≈ 60%
polblogs	5	1,050	2,661	≈ 150%
company	5	336	782	≈ 130%

Table 5.5: G-Tries: Comparison between the sequential version and the parallel version with one core.

5.3 Static Division Approach

In this approach, the graph vertices are divided in a round robin way between workers and each worker processes its portion. In fact the results were not good in general, because this approach depends on the balance in the graph, which means it will be perfect if the amount of work under each vertex is equal, and that wont happen in practice. Usually the work is not balanced and some times it may be up to 20% of the occurrences under one vertex, in this case the speedup will not exceed 5 even if the number of cores equals the number of vertices. The following sections show the speedups obtained by applying this approach on our case study algorithms.

5.3.1 ESU - Speedups

In order to see the scalability of this approach, the speedups of applying it on ESU algorithm are shown in table 5.6. As shown, the speedups we got were between 3.3 and 12.5.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.6**	**2.6**	**4.3**	**8.5**	**6.6**	**7.7**
foldoc	5	**1.7**	**2.9**	**4.5**	**5.9**	**7.4**	**8.3**
septemper11	4	**1.8**	**3.3**	**5.1**	**7.3**	**9.0**	**10.8**
gnutella	6	**1.8**	**2.7**	**3.3**	**4.6**	**4.9**	**5.6**
social	6	**1.6**	**3.0**	**4.7**	**7.3**	**6.8**	**12.5**
wikivote	4	**1.0**	**1.7**	**2.8**	**3.7**	**4.9**	**5.5**
company	5	**1.0**	**1.5**	**2.2**	**2.9**	**3.3**	**3.6**
power	7	**1.0**	**1.8**	**3.0**	**4.8**	**5.4**	**5.1**
neural	6	**1.3**	**1.9**	**1.8**	**2.4**	**2.5**	**3.3**
metabolic	6	**1.9**	**3.4**	**7.7**	**15.3**	**17.1**	**11.0**

Table 5.6: `ESU`: speedups obtained by applying `Static Division` approach.

5.3.2 G-Tries - Speedups

Table 5.7 shows the speedups obtained by applying the static division approach on G-Tries algorithm, speedups are between 2 and 17. The variation of the speedups is high from one graph to another, because this approach depends on the distribution of the subgraphs in the tested graph. Because of that we can see good speedups in some cases as `gnutella` and some very low speedups like in `facebook`. As a result, this approach has some weaknesses that will be solved in the second approach.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.5**	**2.4**	**4.1**	**6.1**	**6.9**	**7.0**
foldoc	5	**1.5**	**1.8**	**2.0**	**2.3**	**2.5**	**2.4**
september11	4	**1.9**	**3.3**	**4.4**	**5.5**	**6.9**	**7.5**
company	5	**1.1**	**1.1**	**1.1**	**1.2**	**1.1**	**1.1**
gnutella	6	**1.8**	**3.6**	**5.5**	**9.4**	**11.3**	**16.6**
facebook	5	**1.1**	**1.9**	**1.9**	**2.0**	**2.0**	**1.9**
netscience	6	**1.7**	**2.6**	**3.6**	**4.1**	**4.5**	**5.2**
wikivote	4	**1.9**	**2.9**	**4.7**	**7.7**	**9.1**	**8.7**
neural	7	**1.3**	**1.9**	**2.1**	**2.6**	**2.5**	**3.1**
metabolic	5	**1.3**	**1.4**	**1.3**	**1.4**	**1.4**	**1.5**

Table 5.7: `G-Tries`: speedups obtained by applying `Static Division` approach.

5.4 Iterative with Time-Limit Approach

This approach was developed to solve the problems of the Static Division one, by using iterative MapReduce depending on a time limit (threshold) to stop workers, collect partial results and redivide the work. Furthermore, for the purpose of maximizing the speedups, the threshold should change from one graph to another. In order to decide what should be the threshold, we did many tests and chose the values that in general maximized the speedups. As a result the initial threshold is being set according to the following formula: $threshold = "(numNodes^2) * (motifSize^3)" nanoseconds$, as could be seen it depends on the graph size and the searched subgraph's size. After the first iteration, the threshold expresses two cases fixed and dynamic threshold, which means either it is going to keep the same value for all the iterations, or it will be readjusted. The following sections show the results we got using each of them.

5.4.1 Fixed threshold

The results obtained were better than the ones from the first approach, as could be seen in the speedups tables in the following sections. However, the problem of this fixed threshold is that when the its value is high many workers are going idle, and when the threshold is low, a lot of time is being spent in saving the states of the workers and redividing the work.

5.4.1.1 ESU - Speedups

Speedups are strongly affected by the threshold and in general we got higher speedups when it was lower, because the time spent in reducers and the time required to save the state and continue work from a saved state is optimized. As shown in table 5.8, the results were better than the ones obtained using the static division approach. For instance, in the first example *polblog*, the speedups increased from to 7.7 to 19.2, and in *neural* graph the speedups increased from 3.1 to 16.2 which is a good improvement.

5.4.1.2 G-Tries - Speedups

The speedups obtained by applying this approach on G-Tries were better than on ESU as table 5.9 shows. The average speedups obtained was 28.6. However, as shown in the table some times the speedups are around 40, and sometimes 16. This difference in the achieved speedups is mainly because of the effect of the threshold, since the initial value is not always suitable for the specific case we are computing.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.9**	**3.6**	**6.1**	**10.3**	**12.5**	**19.2**
foldoc	5	**1.6**	**3.1**	**5.2**	**7.2**	**10.3**	**12.9**
facebook	5	**1.9**	**3.5**	**4.0**	**6.8**	**9.6**	**12.5**
company	5	**1.7**	**3.0**	**5.1**	**7.1**	**9.2**	**11.7**
september11	4	**1.7**	**3.2**	**4.7**	**7.4**	**12.1**	**17.3**
wikivote	4	**1.8**	**3.2**	**5.7**	**8.1**	**12.4**	**14.9**
netscience	7	**1.9**	**3.4**	**5.7**	**7.7**	**10.0**	**9.2**
gnutella	6	**1.7**	**2.9**	**4.3**	**5.6**	**6.4**	**8.7**
neural	6	**1.9**	**3.4**	**5.3**	**8.1**	**13.4**	**16.2**
metabolic	6	**1.9**	**3.3**	**5.7**	**9.8**	**16.4**	**17.7**

Table 5.8: `ESU`: speedups for ESU-Iterative with fixed Threshold.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.6**	**3.8**	**8.2**	**15.4**	**26.0**	**36.1**
foldoc	5	**1.9**	**3.8**	**7.2**	**10.7**	**22.9**	**27.6**
septemper11	4	**1.9**	**3.6**	**7.1**	**12.7**	**19.7**	**25.8**
gnutella	6	**1.9**	**3.8**	**7.1**	**11.2**	**17.1**	**21.3**
company	5	**1.9**	**3.7**	**6.3**	**7.6**	**14.0**	**16.7**
facebook	5	**1.9**	**3.9**	**7.7**	**14.2**	**26.4**	**38.9**
wikivote	4	**1.9**	**3.9**	**7.3**	**13.2**	**23.0**	**33.5**
neural	7	**1.9**	**3.9**	**7.5**	**14.3**	**27.8**	**41.4**
metabolic	6	**2.0**	**4.0**	**7.2**	**13.0**	**18.1**	**22.6**
netscience	9	**1.8**	**3.9**	**6.3**	**10.3**	**17.3**	**21.7**

Table 5.9: `G-Tries`: speedups for G-Tries-Iterative with fixed Threshold.

5.4.2 Dynamic threshold

The best results were obtained by applying this approach which adjusts the threshold during the computation as follows. First the workers start computation with an initial threshold value equal to the one in the fixed threshold:
$threshold = "(numNodes^2) * (motifSize^3)"nanoseconds.$
Then after every iteration the threshold is readjusted according to the number of workers who waited in the previous iteration and how much time they waited. There are two cases, either non of the workers wait, or some of them do. In case non of the workers went idle in the previous iteration we increase the threshold by 20%, in order to choose the amount of time that should be increased, we did tests on three different networks, figure 5.1 shows the results we got, our goal was to maximize the speedup. Because of that, we chose the percentage 20% to be the used one. On the contrary, if some workers went idle during the previous iteration, we decrease the threshold by an

amount equal to:

$$newT = oldT - ((oldT * numIdleWorkers)/totNumWorkers) - (0.2 * avgWaittedTime).$$

We got this formula by supposing that it should depend on the previous information which are, the number of workers who went idle and waited ($numIdleWorkers$), and the average amount of time that each worker waited ($avgWaittedTime$). Then in order to decide the percentages, we did many tests and chose the formula that maximized the speedups for the majority of the tested graphs.
We should reassert that in our machine each pair of cores shares its 2MB L2 and 64KB L1 instruction cache. This makes it harder to obtain better speedup because these cores are not completely independent. However, the results we obtained are promising, as could be seen in the next sections. The speedups from G-Tries algorithm were better than the ones from ESU, that is because saving the occurrences in the trie is proposing an overhead in ESU, and this problem does not exist in G-Tries since we know the subgraphs beforehand.

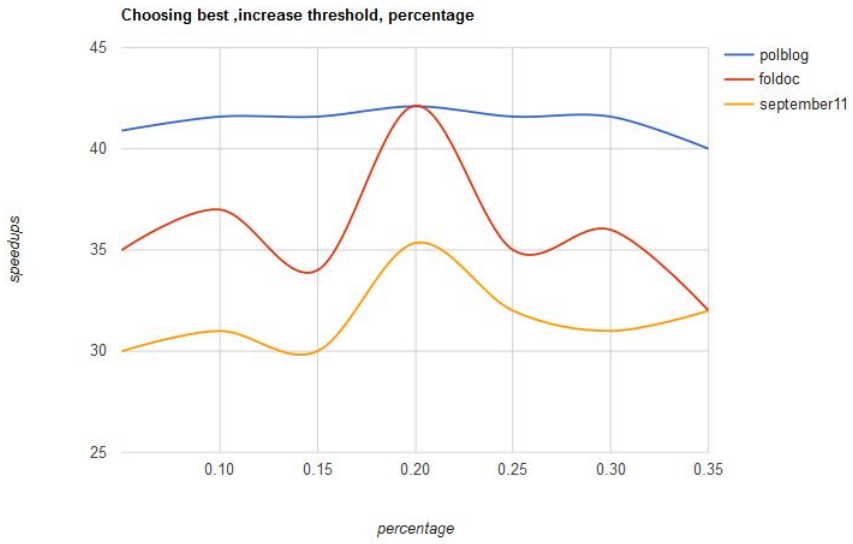

Figure 5.1: Choosing the best percentage to increase the threshold.

Table 5.10 shows an example that describes how the threshold is changing after each iteration, each line in the table represents an iteration, and the needed information to change the threshold is displayed.

Iteration	Threshold (s)	Total number of workers	# workers who waited	average waited time (s)
1	4.4	64	0	0
2	5.3	64	0	0
3	6.3	64	0	0
4	7.6	64	8	1.4
5	6.4	64	33	1.7
6	2.8	64	58	1.4
7	0.1	64	54	0.08
8	0.1	64	61	0.09

Table 5.10: Example about the changing of the threshold from one iteration to the next.

5.4.2.1 ESU - Speedups

We were able to obtain speedups up to 23.4 by applying our final approach on ESU algorithm, the results are better than the ones ones obtained using the previous approaches. Table 5.11 expresses the obtained speedups.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.9**	**3.5**	**6.1**	**10.4**	**17.1**	**23.4**
foldoc	5	**1.6**	**3.2**	**5.0**	**7.4**	**10.6**	**13.7**
septemper11	4	**1.8**	**3.1**	**5.2**	**8.5**	**12.7**	**18.3**
gnutella	6	**1.7**	**3.0**	**4.1**	**5.8**	**7.3**	**9.5**
wikivote	4	**1.7**	**3.1**	**5.4**	**8.3**	**11.5**	**12.4**
facebook	5	**1.8**	**3.9**	**5.0**	**10.4**	**19.4**	**21.0**
company	5	**1.8**	**3.2**	**5.1**	**8.1**	**10.3**	**12.4**
neural	6	**1.9**	**3.2**	**6.5**	**9.44**	**15.7**	**16.5**
metabolic	6	**1.9**	**3.9**	**7.2**	**11.6**	**17.9**	**20.5**
netscience	7	**1.9**	**2.5**	**5.1**	**8.5**	**10.0**	**16.0**

Table 5.11: ESU: speedups for ESU-Iterative with dynamic Threshold.

Directed subgraphs

Table 5.12 shows that our strategy is general and could be applied when searching directed subgraphs, since the results were similar to the ones obtained before when searching for undirected subgraphs.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
company	5	**1.7**	**3.1**	**5.3**	**7.8**	**10.5**	**11.1**
septemper11	4	**1.6**	**3.2**	**4.7**	**7.9**	**13.0**	**17.9**
wikivote	4	**1.8**	**3.1**	**5.2**	**8.1**	**11.6**	**12.7**

Table 5.12: `ESU`: Speedups for ESU-Iterative with dynamic Threshold when searching for directed subgraphs.

5.4.2.2 G-Tries - Speedups

This section shows the promising results we got by applying our strategy on G-Tries algorithm. We achieved 92.5% efficiency using 32 cores, also with 64 cores we still achieve speedups up to 45.5.

Table 5.13 expresses the obtained speedups.

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
polblog	5	**1.9**	**3.6**	**7.2**	**13.6**	**26.0**	**41.8**
foldoc	5	**1.9**	**3.6**	**7.1**	**14.2**	**26.4**	**42.1**
septemper11	4	**1.9**	**3.7**	**6.9**	**12.7**	**24.1**	**36.0**
gnutella	6	**1.9**	**3.8**	**6.7**	**12.5**	**24.3**	**40.1**
company	5	**1.9**	**3.8**	**7.1**	**12.2**	**20.1**	**25.2**
facebook	5	**1.8**	**3.4**	**7.1**	**13.9**	**25.4**	**36.6**
wikivote	4	**1.9**	**3.8**	**7.1**	**12.5**	**20.6**	**25.1**
neural	7	**1.9**	**3.7**	**7.3**	**13.9**	**26.1**	**43.2**
metabolic	6	**1.9**	**3.6**	**7.1**	**13.9**	**26.6**	**40.0**
netscience	9	**1.9**	**3.9**	**6.8**	**11.3**	**19.0**	**25.9**

Table 5.13: `G-Tries`: speedups for G-Tries-Iterative with dynamic Threshold.

Directed subgraphs

The previous tests were done by searching for undirected subgraphs. However, for the purpose of showing that our strategy is general and could be applied when searching directed subgraphs, the following test were done. Table 5.14 shows that speedups were very similar to the ones obtained before.

Figure 5.2 shows the relation between the required computation time and the number of workers, showing the difference between the parallel and sequential versions. As shown in the figure, with one core the parallel version is slower because of the overhead, but with two and more cores it becomes much faster until reaching a speedup of 45.5

Network	subgraph size	#workers: speedup					
		2	4	8	16	32	64
foldoc	5	**1.9**	**3.4**	**5.4**	**13.0**	**26.1**	**43.6**
company	5	**1.8**	**3.8**	**6.6**	**13.5**	**23.3**	**31.4**
wikivote	4	**1.9**	**3.4**	**5.4**	**13.0**	**29.6**	**45.5**

Table 5.14: `G-Tries`: speedups for G-Tries-Iterative with dynamic Threshold when searching for directed subgraphs.

using 64 cores. In this figure, the vertical access represents the computation time in seconds and the horizontal access represents the number of cores. Furthermore, with such speedups, some examples which were very hard to run will be possible and that gives the user the ability to search for larger subgraphs, which is very important toward extracting new information from that network. For instance, we run a test on *wikivote* network to find subgraphs of size 5 using the sequential version, it took around two days (47 hours); However, using our parallel algorithm and using 64 workers it took 1.2 hours.

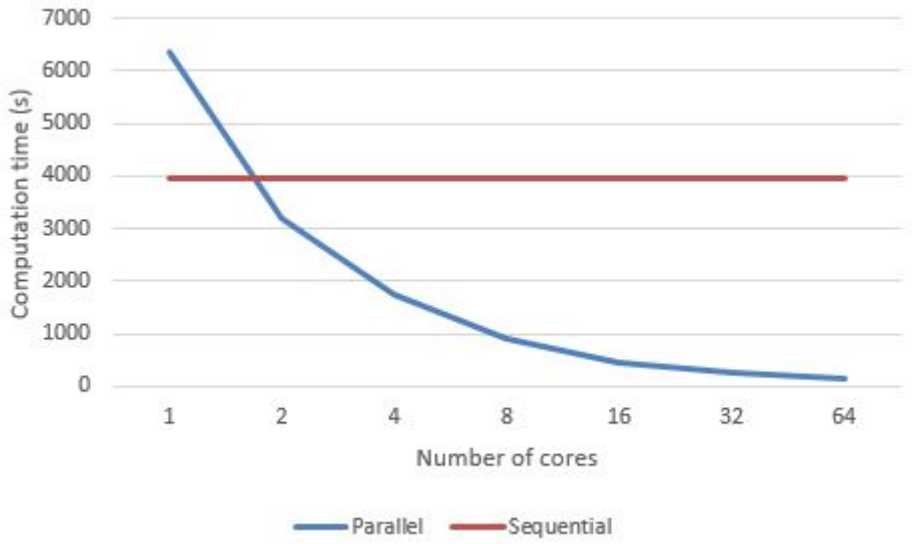

Figure 5.2: `G-Tries`: Sequential vs Parallel needed computation time for the *polblog* network with $k = 5$.

Figure 5.3 summarizes our results for g-tries iterative MapReduce using dynamic thresholding, by plotting the average speedup of all tested networks in both the directed and undirect cases. We can clearly see that we achieve close to linear speedup up to 32 cores and even in the 64 cores case we obtain considerable speedup only limited by the used hardware. Furthermore, detailed analysis of the results shows that the better speedups are obtained for the cases in which the computation time is higher.

This means that we are cutting the needed time in the use cases that most need it and means that we could potentially scale well for bigger cases. For instance, in the *company* network, the computation time for 32 cores is already only 39 seconds and it is hard to improve upon this. By contrast, in the *neural* network, 32 cores take 506 seconds and so there is still room for parallel improvement when doubling the number of processors.

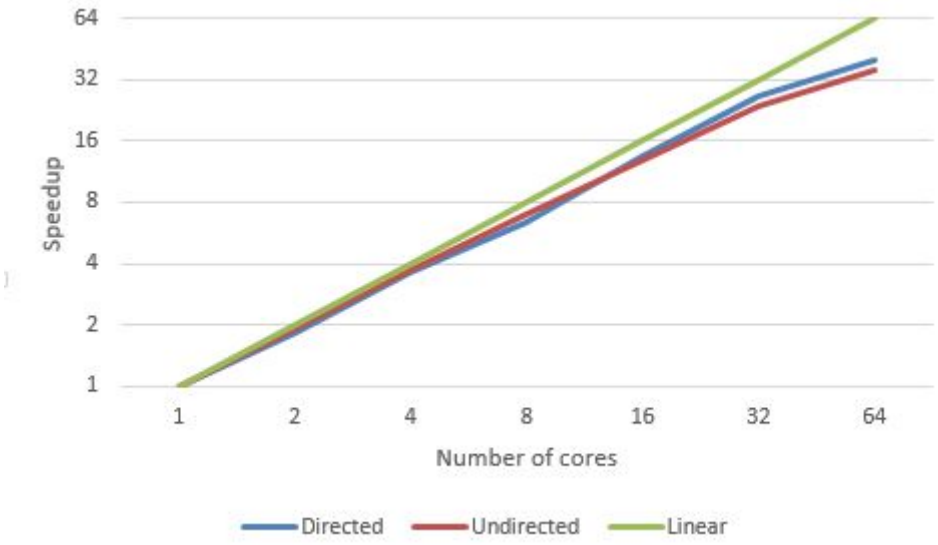

Figure 5.3: `G-Tries`: Average speedup for all tested networks.

5.4.3 Comparison with competing algorithms

In section 3.3 we mentioned different parallel algorithms in the area of subgraph search. Since most of the mentioned algorithms do not parallelize the same algorithms, it was not possible to compare our results with theirs; However, our strategy is different and does not have the weaknesses of the large intermediate data nor the memory issues that they have.

The most recent algorithm of the mentioned ones do parallelize an algorithm that we are parallelizing using MapReduce [VKK15]. In order to compare our results with theirs, and since they do not have their code online, we use the table of results in their paper. Their second best speedup was using a protein protein interaction (PPI) network which has **2,365** nodes, and searching for subgraphs of size **7** in this network. The processors in our machine is slower, ours are *2.3 GHz* and theirs are *3.4 GHz*. However, we use the same number of cores toward making the comparison as close as possible. Table 5.15 shows the obtained results, As shown in the table, our sequential

ESU is faster than theirs, and our speedups and efficiency are higher. Furthermore, G-Tries is 16 times faster using one processor. Moreover, our speedup and efficiency are much higher as shown in the table. As conclusion, to do this test using approximately the same machine, their parallel algorithm took **5.5 hours**, our parallel ESU took **2.2 hours** and our parallel G-Tries took **4.4 minutes**.

Method	**Sequential time**	**Parallel time**	**Speedups**	**Efficiency**
`competitor ESU`	172,800	19,686	8.78	$\approx 15.7\%$
`our ESU`	142,950	7,854	18.20	$\approx 32\%$
`our G-Tries`	9,539	266	35.86	$\approx 64\%$

Table 5.15: Comparison between our algorithms and a recent competing one.

Cytoscape Plugin 6

In this chapter we present our plugin inside the `Cytoscape` software, which was developed toward making the subgraph search algorithms more user friendly. First, we express the characteristics of `Cytoscape`. Then we explain the features of our plugin and how it works showing some diagrams and screen-shots.

6.1 Motivation

Cytoscape is an open source software platform for large-scale the visualizing and analysis of complex networks, and integrating these networks with annotations, gene expression profiles and other state data [Cyt02b]. It was originally created at the Institute of Systems Biology in Seattle in 2002. Now, it is developed by an international consortium of open source developers. The current version, 3.2.1, was released in February 2015.

Cytoscape provides a set of features for data integration, analysis and visualization. It supports the visualization of nodes and edges as a two dimensional network using a variety of layout algorithms. Hierarchical layout, spring-embedded layout, and circular layout are examples of the supported layouts [SMO+03]. Moreover, it supports a wide variety of visual properties in order to control the appearance of the nodes and edges, for instance node color, shape and size; edge color, thickness and style.

Additionally, more specialized algorithms are added to Cytoscape as plugins. Plugins are given access to the core network model and can control the network display [SMO+03]. Furthermore, plugins could be developed by anyone by using the open Cytoscape API which is based on java. Most of the plugins are freely available in the Cytoscape applications store [Cyt02a].

6.2 State of the art

There is not any plugin in Cytoscape application store that does the motif discovery in the same way we do. However, the following plugins do similar functions:

6.2.1 NetMatch

NetMatch was developed in 2011, it finds user defined network motifs [SBS+11]. The user can choose a specific motif from a list of motifs, and the plugin will count the occurrences of this specific motif, unlike our plugin which counts all the k-size motifs. NetMatch works only with two old versions of cytoscape which are **2.7 and 2.8.1**, knowing that the current Cytoscape version is **3.4.0**.

6.2.2 CytoKavosh

CytoKavosh plug-in was developed in 2012, it uses Kavosh algorithm for finding network motifs and is based on counting all k-size sub-graphs of a given network graph (directed or undirected) [MNAR+12]. However, they only support Linux OS, and the old versions of Cytoscape. CytoKavosh does not work with windows OS neither with **3.0.0** and newer versions of Cytoscape. The main difference between CytoKavosh and our plugin is the algorithm used in counting motifs, we use G-Tries which is in average 25 times faster than Kavosh. Moreover, we have more visualization features that will be explained next.

6.2.3 GraphletCounter

It computes the graphlet signatures of individual nodes or of motifs, which can be specified by files generated by the motif-finding tool mfinder [WS11]. Although graphlet and motif are similar concepts, the difference between them is that random networks are not used to verify over-representation in graphlets. GraphletCounter plugin displays graphlet signatures visually within Cytoscape, and can output graphlet data for integration with larger work-flows. Equally to the previous plugins, it does not work with newer versions of Cytoscape.

6.3 Motif Discovery Plugin

In order to satisfy the needs of users in doing network motif discovery (subgraph search) and take the advantages of existing network visualization software like Cytoscape, we developed our user friendly, easy to use and fast plugin.

This plugin uses the current fastest motif discovery algorithm which is G-Tries, which makes it very fast.

Generally speaking, after installing the plugin, the user specifies the required parameters (motif size, motif type, number of random networks), and the plugin will execute the G-Trie algorithm on the current network using the specified parameters and then visualize the result using the visualization power of Cytoscape.

The current version of the plugin allows its user to do the following:

- **Load his network into the Cytoscape environment**: this is the first step that the user does, he can either draw the network by adding nodes and edges or he can load it from a file. In addition, it is not a problem if the nodes' numbers in the graph file start from zero or one or any number.
- **Specify the size of the motif**: the current version allows the user to count motifs of sizes between **3** and **9**.
- **Specify the motif type**: the user can count **directed** or **undirected** subgraphs.
- **Specify the number of random networks**: in order to measure the statistical significance for the network (Z-Score).
- **Run the algorithm**: after loading the network and choosing the required parameters, the user can run the algorithm by pressing the run button.
- **See the work progress**: while the algorithm is running, there will be a progress bar that shows how far along it is in the process. Moreover, there is a label that tells the expected remaining time. For this purpose we run a sampling algorithm to expect the number of occurrences, this will take a very small amount of time before we run the algorithm, and after having it we run the required algorithm and set the progress bar depending on the current number of occurrences and the expected one.
- **See the results in a friendly way**: the results are shown in a table of three columns (motif representation , number of occurrences, Z-Score).

- **Change the order of the subgraphs**: rows in the result table could be ordered depending on any of the last two columns (descending or ascending).
- **Select any of the table rows and the plugin will draw the corresponding subgraph**: because it is hard for the user to know the subgraph according to its binary representation, especially when the size is large, we developed an algorithm that draws the subgraph which correspond to the selected raw. The subgraph is drawn in the same working area toward not confusing the user with new windows.
- **Select any of the table rows and the plugin will color the network according to the occurrences of this subgraph**: this is one of the most important features which uses the Cytoscape node's color attribute. We color the nodes according to the number of times each one occurred in the selected subgraph, for instance if there are two occurrences for subgraph of size 3 : (1,2,3)(1,4,5) then node 1 appeared two times and the other nodes appeared only once, in this case the color of node 1 will be different from the others. In the current version we are coloring the nodes with maximum ten colors, in other words we divide the range of number of appearances into ten ranges, each range will be colored in the same color, the used colors are shown and ordered in the interface.
- **See the range of frequencies that each color represents**: by clicking on any of the ten colors, in the plugin interface, it will tell the user the range of frequencies that it represents.

6.4 Plugin Design

6.4.1 Use Case Diagram

Figure 6.1 shows an outside view of the system through a use case diagram. It represents the interaction between our unique actor (user) and the different use cases in which the user is involved.

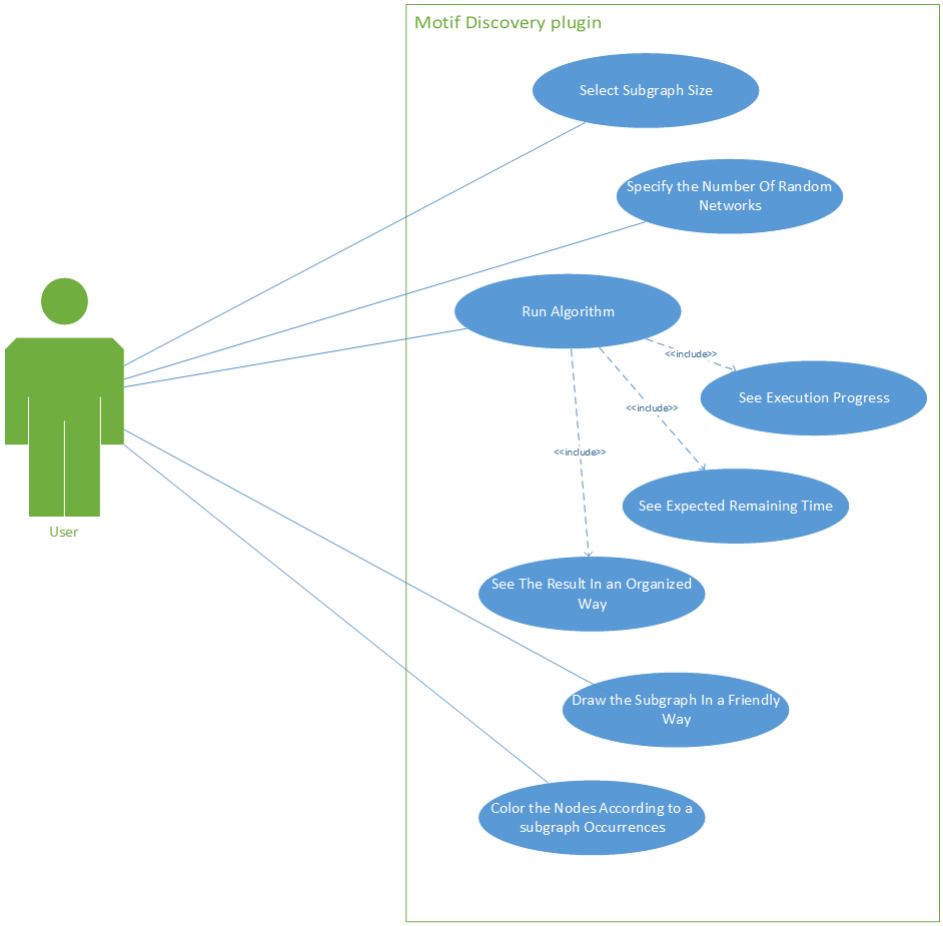

Figure 6.1: Use Case Diagram for our plugin.

6.4.2 Sequence Diagram

As an illustration, figure 6.2 is a sequence diagram that shows object interactions arranged in time sequence. It depicts components involved in the scenario and the sequence of messages exchanged between them to carry out the functionality of the scenario.

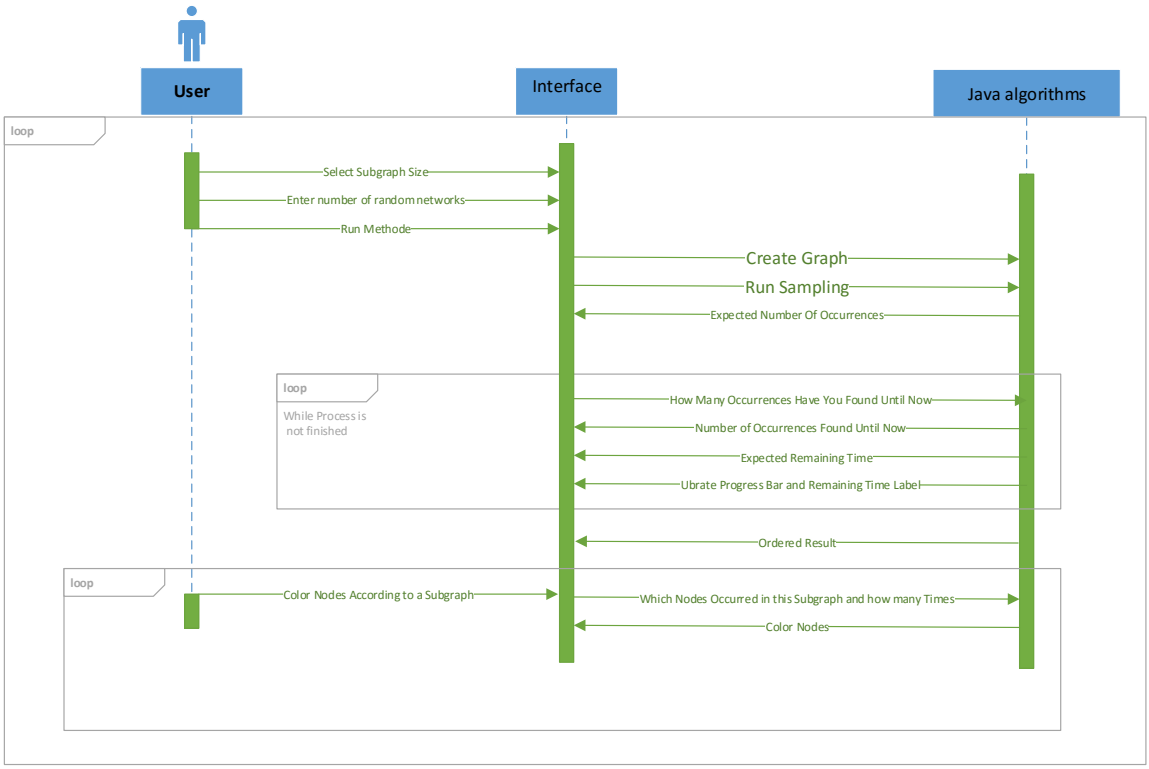

Figure 6.2: Sequence Diagram for our plugin.

6.5 GUI-Snapshots

6.5.1 Running

Figures 6.3 shows how the system looks like while the algorithm is running. The user can see the work progress and the expected remaining time.

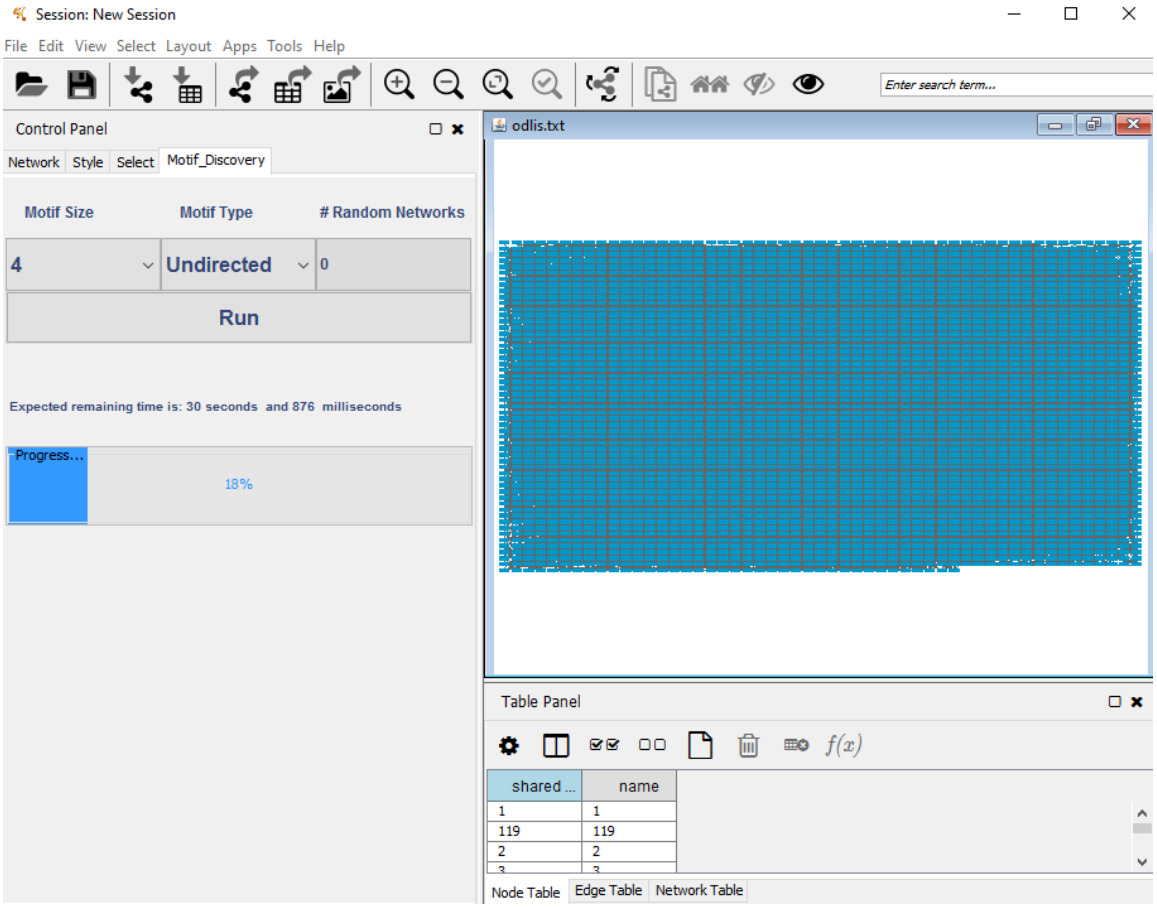

Figure 6.3: Motif-Discovery: Running the work

6.5.2 showing result

In figure 6.4, the work is done and the result is displayed. A summary about the work could be seen below the run button, it includes the number of types of subgraphs found, the number of occurrences and the computation time. Moreover, the table shows the details about each subgraph type. The small subgraph above the table visualizes the subgraph that the user selected, to the right of this subgraph is the color set used, the light green will be the color representing the least frequent nodes and the dark red is the color of the most appeared ones. Moreover, when the user clicks any of those

colors, a message box will tell him where it appears as show in the figure. As shown in the work place the nodes are colored according to their occurrences in the selected subgraph type, the numbers attached with these colors are shown in the table below the work space in the column *#occurrences*, in this table each row represents a node.

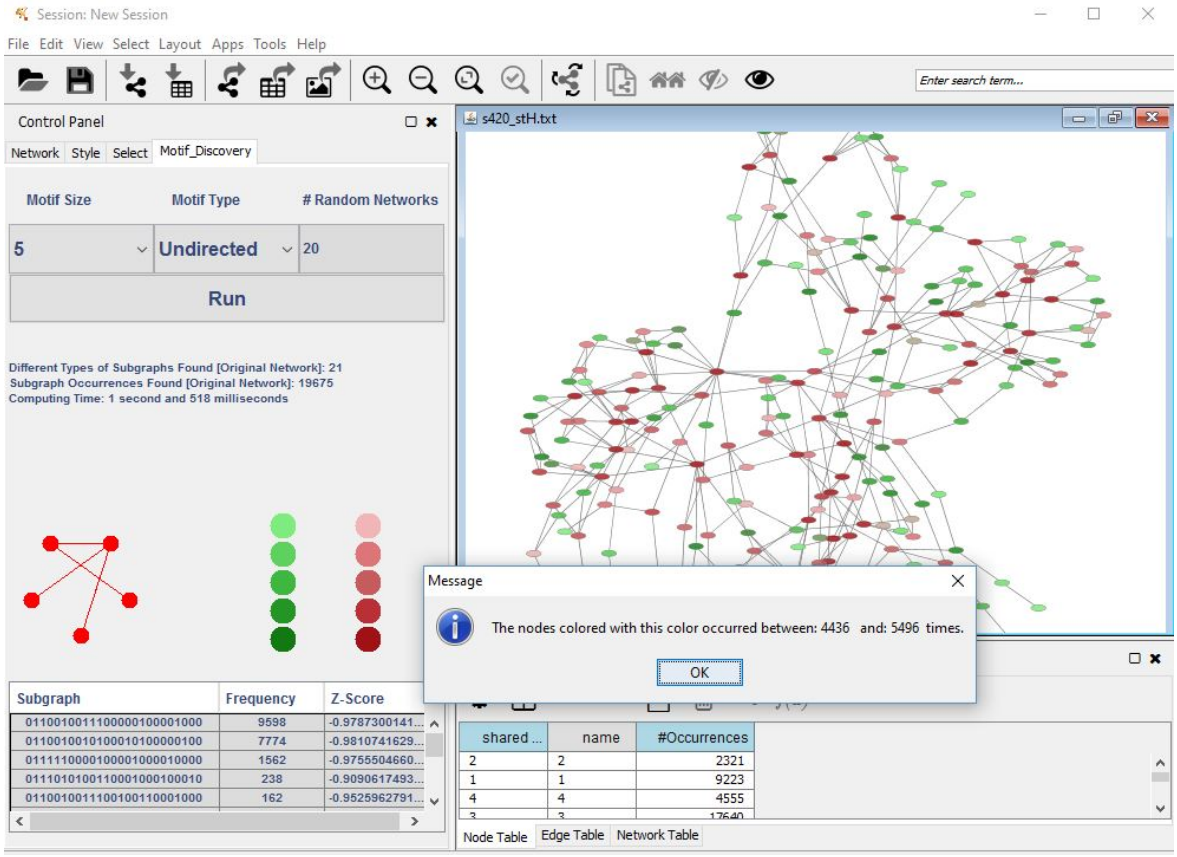

Figure 6.4: Motif-Discovery: Displaying the result

Furthermore, the are adding the number of occurrences as new property of the node, and that will allow the user to use it in different visualization functions, for instance he could adjust the layout depending on this new property as shown in figure 6.5. Also it could be exported from cytoscape as a *csv* file.

After all, the plugin is now available in *Cytoscape app store* and could be downloaded and directly used [eR16], it does not depend on any library unlike the first version that is explained in appendix A. Moreover, it works with any operating system, we testes it on Windows, Mac and Linux.

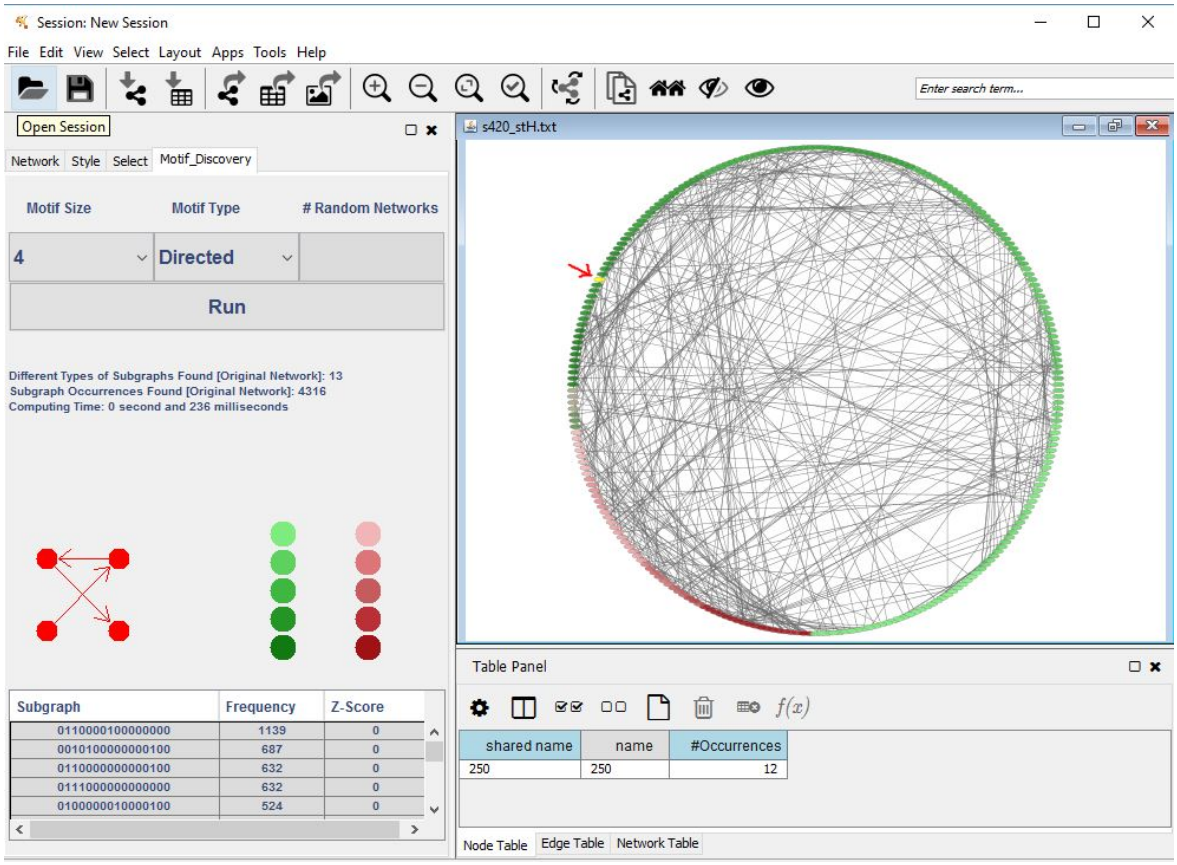

Figure 6.5: Motif-Discovery: using the number of occurrences as a new property.

Conclusions and Future Work 7

Complex networks are used in a wide range of artificial and natural systems. The detection of small patterns in these networks lead to a better understanding of their structure and functionality. This operation is called subgraph search and has been applied to networks in many fields. However, it is a computationally hard problem and because of that its application is limited by the size of the pattern being searched and the size of the network. For the purpose of decreasing those limitations, this work develops a parallel MapReduce strategy that speeds up subgraph census in complex networks. Moreover, a plugin to do subgraph search in a friendly way was built inside *Cytoscape* software. This chapter summarizes the main contributions done and concludes with a directions for future work.

7.1 Contributions

The main goal of this thesis is to large scale the **Subgraph Search** and make it available to users in a friendly way. The following points are the main contributions of our work.

- **Java implementation of ESU and G-Tries**: Since the technologies we are using do not support native languages, and our case study algorithms (ESU, G-Tries) are implemented using C++, our first contribution of the thesis is a java implementation for the sequential versions of those algorithms.
- **Delay ismorphism tests in ESU:** For the purpose of making ESU faster, the isomorphism test were delayed to the end of the enumeration operation, instead of doing them after finding each occurrence. This results in reducing the required number of tests which are time consuming.
- **Map-Reduce Strategy for unbalanced "tree-like" parallel search:**

  This is the major contribution. We designed and implemented a general scalable parallel MapReduce strategy for subgraph search treelike algorithms. Our

implementation was done inside Spark framework, which is widely available on cloud computing providers such Amazon Web Services, and is also available on different operations systems.

  - **Strategy application in both ESU and G-Trie**

    We apply the strategy on both ESU and G-Tries algorithms, which are different, ESU is network-centric algorithm and G-Tries is set-centric one. Since MapReduce was not originally developed for this kind of data, so in order to do load balancing between the workers, we dynamically divide the search tree among them, we developed an efficient sharing mechanism that is able to stop, save the remaining work in a collection of work sets and resume the execution. By being able to successfully apply our strategy to two different algorithms we also display its generality.

  - **Thorough empirical analysis** we test the scalability of our strategy by doing an extensive experimental evaluation for the implementation of both algorithms. We tested them using large set of representative networks from different fields. Speedups up to 45 were achieved using a machine with 64 cores.

    To the best of our knowledge, there is not any MapReduce for set-centric subgraph search algorithms, so ours is the fastest available MapReduce one. Regarding the network-centric algorithms, our parallel ESU performs better than the existing MapReduce approach. As a result, our strategy expands the limits of subgraph counting applicability, allowing an exploration of larger subgraphs in bigger networks.

- **Motif Discovery Plugin**

  we developed a plugin for subgraph census in Cytoscape software, the goal of this plugin is using the visualization power of Cytoscape to make the subgraph search more friendly for the user, especially people who are not from computer science and not familiar with command line. The plugin was published in the application market of Cytoscape, and now any user can use it by simply downloading it from the market.

7.2 Future Work

There are still many areas that deserve improvement, either in the strategy itself in order to achieve better speedups, or by adding more features to the plugin to advance its usage. Next we mention some points for future research.

- **Intelligent automatic way in adjusting the threshold:** Since the main factor affecting the speedups, is adjusting the threshold, we would like to study how this process can be automated in a more intelligent way toward achieving better speedups.
- **Supporting larger networks by applying the strategy on other algorithms that do not use adjacency matrix:** a major limitation of ESU and G-Tries algorithms is assuming that the entire network can fit in the main memory. This limits the applicability to relatively small networks. And nowadays networks with billions of nodes are becoming widely available. Therefore, we would like to apply our strategy on algorithms that does not use adjacency matrix.
- **Apply the strategy on sampling algorithms** which give an approximation about the number of occurrences; we would like to parallelize an efficient sampling algorithm using our strategy. Moreover, this sampling could help in knowing the perfect way in adjusting the threshold for the network being processed, that is by running the sampling before the exact enumeration.
- **Real world scenarios** Practically speaking, we would like to use the advantages of the scalability of our algorithm in analyzing real world networks, by searching for new subgraph patterns that can lead to deeper understanding of their structure.
- **Making the code public** In order to let all practitioners benefit from our scalable algorithm we are planing to make our code public, so that any person can use it either in his machine or in a computing cloud.
- **Publish our work by submit it to conference or journal:** because of time constrains we have not managed to submit our work to be published yet, but we are going to do that.
- **Gephi version of the plugin** For the purpose of letting more people count subgraphs in a fast and friendly way, we would like to do another plugin inside

Gephi software which is widely used as *Cytoscape*. In fact, those two software are widely used by users who study networks.

- **Add more feature to the plugin** In this work, we presented the first version of our plugin, which we plan to continue improving. For instance, we will parallelize the algorithm using *pthreads* to take the advantages of multiple cores in the user's machine. Also, we will study the possibility of mining the users' results if they permit, and do researches on that data from all over the world.

7.3 Closing Remarks

We feel that the main objectives of this thesis were accomplished. A general MapReduce strategy was developed and applied on two different algorithms. Moreover, a user friendly plugin was developed and published. On the other hand, for the author, researching subgraph search and parallelization approaches in addition to working with open source software (Cytoscape), have been very interesting, and contributed in developing his programming and research skills.

Cytoscape Plugin using JNI A

Motif-Discovery plugin was developed in the same time with our parallel strategy, so before having the java implementation of G-Tries, we used the existing C++ code of the algorithm and because *Cytoscape API* is based on Java we used JNI (Java Native Interface) to make the bridge between the java interface and the C++ algorithm, more details about the version that uses JNI could be found in the appendix A.
However, after having the Java implementation, we decided to use it, in order not to force the user to install external libraries, for the purpose of making the installation and usage of the plugin easy and do not depend on any external software or library.

A.0.1 What is JNI

JNI (Java Native Interface) is a powerful feature that allows taking the advantage of the java platform, and still use the code written in native languages. JNI is a two way interface, it permits the java application to call native code and vice versa. Using JNI, java application invokes the native methods in the same way as it is invoking methods that are implemented in java [Lia99]. As shown in figure A.1, writing and running a program with the use of JNI needs 6 main steps.

A.1 Count-Occurrences Plugin

This plugin supports three different algorithms for motif discovery which are G-Tries, ESU and Subgraph. Since we started developing the plugin before having our java implementation for the mentioned algorithms, we needed to use the available c++ implementation which is very optimized. However, the Cytoscape API is based on Java, for that reason we used JNI (Java Native Interface) to make the bridge between the Java interface and the C++ algorithms. Generally speaking, the plugin consists of two main parts, the core and the interface; The core illustrates the native algorithms

and the interface is the java code which uses the visualization power of Cytoscape. Those two parts are connected throw JNI as shown in figure A.2.

The Core

The native algorithm form the engine of the plugin. For the purpose of using the existed C++ code we needed to do the following; After generating the header file of the java interface (step 3 in figure A.1), we implemented the C++ class of this header file, which works like the controller of the core, it has the function that will be invoked from the java interface (native functions), those functions invoke the census algorithms. Moreover, the census algorithms' functions needed to be edited for the purpose of being invoked the way we wanted. After all, we packaged them all together in a dynamic link library (.dll) package.

The Interface

The interface or the java part is the code that will use the core data structure and windows of Cytoscape. It has the controls that the user will use to enter his ... the interface will invoke the native procedures with this information,

A.2 Plugin Design

A.2.1 Sequence Diagram

As an illustration, figure A.3 is a sequence diagram that shows object interactions arranged in time sequence. It depicts components involved in the scenario and the sequence of messages exchanged between them to carry out the functionality of the scenario.

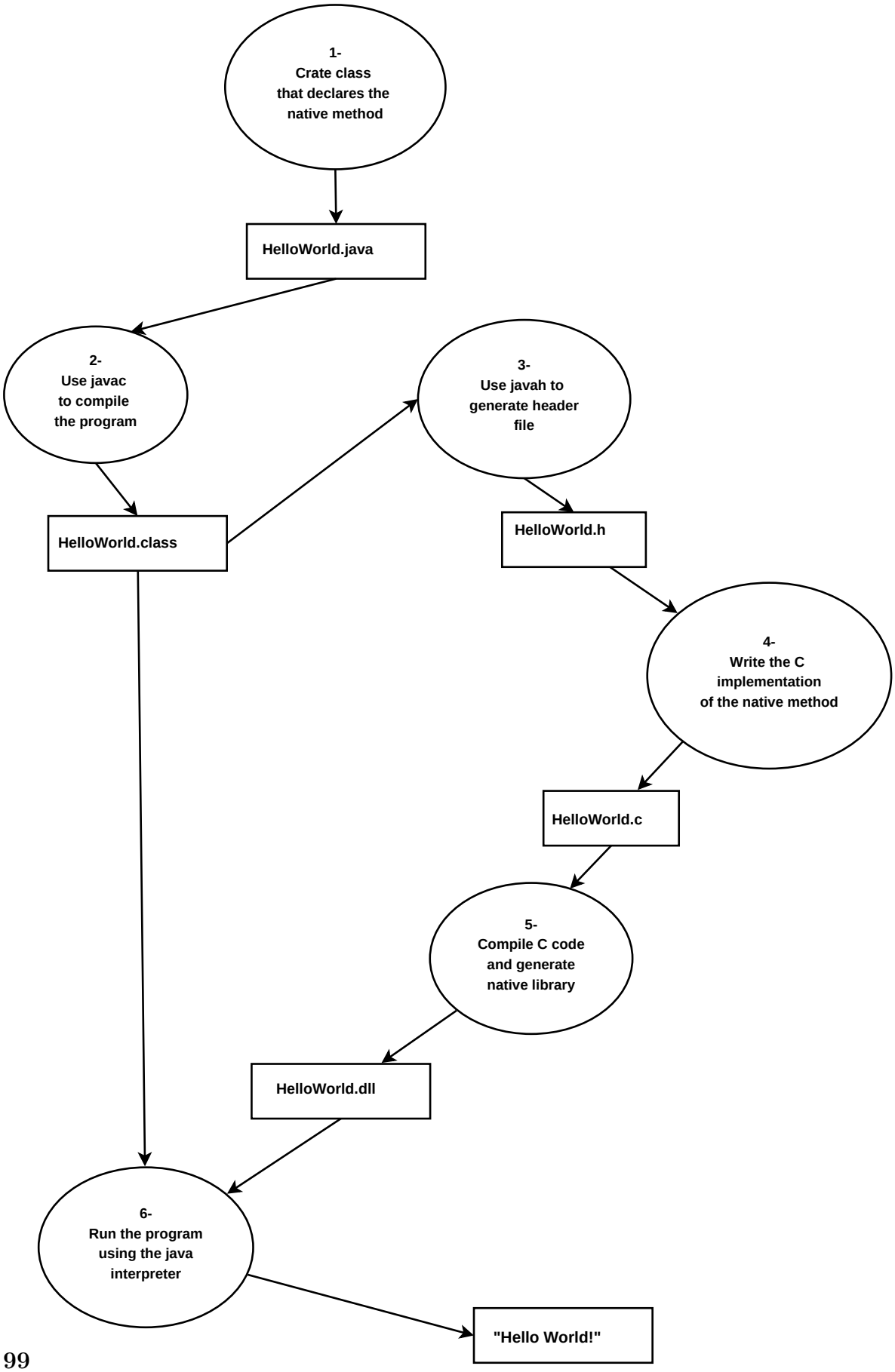

Figure A.1: JNI: steps in writing and running a Hello World program. Adapted from [Lia99]

Figure A.2: Count-Occurrences: the main components of the old version.

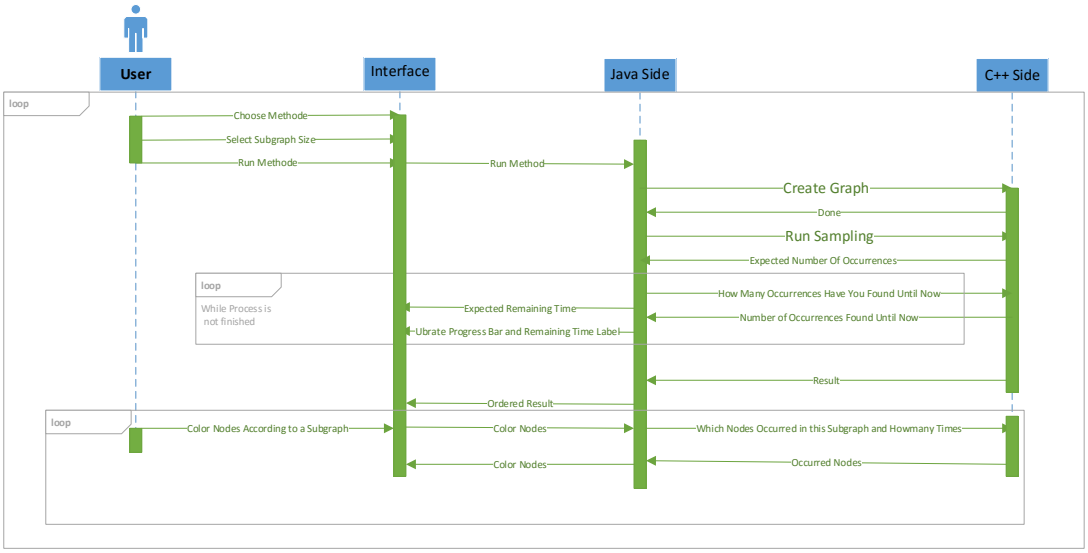

Figure A.3: Sequence Diagram for the old version of our plugin .

References

[AA04] István Albert and Réka Albert. Conserved network motifs allow protein–protein interaction prediction. *Bioinformatics*, 20(18):3346–3352, 2004.

[AFU13] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 62–73. IEEE, 2013.

[AG05] Lada A. Adamic and Natalie Glance. The political blogosphere and the 2004 u.s. election: Divided they blog. In *3rd International Workshop on Link Discovery*, LinkKDD '05, pages 36–43, New York, NY, USA, 2005. ACM.

[Apa16a] Apache. Apache hadoop. `https://hadoop.apache.org/`, 2016.

[Apa16b] Apache. Apache maven project. `http://maven.apache.org/index.html`, 2016.

[Are14] Alex Arenas. Alex arenas website - network datasets. `http://deim.urv.cat/~alexandre.arenas/data/welcome.htm`, 2014.

[ARF16] Miguel Araujo, Pedro Ribeiro, and Christos Faloutsos. Faststep: Scalable boolean matrix decomposition. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 461–473. Springer, 2016.

[ARS14] David Aparicio, Pedro Ribeiro, and Fernando Silva. Parallel subgraph counting for multicore architectures. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE CS, August 2014.

[Bar09] Albert-László Barabási. Scale-free networks: a decade and beyond. *science*, 325(5939):412–413, 2009.

REFERENCES

[Bar16] Albert-Laszlo Barabasi. *Network science*. Cambridge University Press, 2016.

[BGP07] Kim Baskerville, Peter Grassberger, and Maya Paczuski. Graph animals, subgraph sampling, and motif search in large networks. *Physical Review E*, 76(3):036107, 2007.

[BM06] Vladimir Batagelj and Andrej Mrvar. Pajek datasets. `http://vlado.fmf.uni-lj.si/pub/networks/data/`, 2006.

[BMZ02] Vladimir Batagelj, Andrej Mrvar, and Matjaz Zaversnik. *Network analysis of texts*. University of Ljubljana, Inst. of Mathematics, Physics and Mechanics, Department of Theoretical Computer Science, 2002.

[Boy08] Chas Boyd. Data-parallel computing. *Queue*, 6(2):30–39, 2008.

[BS12] Ed Bullmore and Olaf Sporns. The economy of brain network organization. *Nature Reviews Neuroscience*, 13(5):336–349, 2012.

[CFSV04] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[CHLN06] Jin Chen, Wynne Hsu, Mong Li Lee, and See-Kiong Ng. Nemofinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 106–115. ACM, 2006.

[Col16] Smith College. Running cpp programs on hadoop. `http://cs.smith.edu/dftwiki/index.php/Tutorials`, 2016.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[CRBS12] Sarvenaz Choobdar, Pedro Ribeiro, Sylwia Bugla, and Fernando Silva. Comparison of co-authorship networks across scientific fields using motifs. In *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, pages 147–152. IEEE, 2012.

[CRS12] Sarvenaz Choobdar, Pedro Ribeiro, and Fernando Silva. Motif mining in weighted networks. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 210–217. IEEE, 2012.

[CRTVB07] L da F Costa, Francisco A Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007.

[Cyt02a] Cytoscape. Cytoscape app store. `http://apps.cytoscape.org/`, 2002.

[Cyt02b] Cytoscape. Cytoscape software. `http://cytoscape.org/`, 2002.

[DA05] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.

[DDLMM13] Manlio De Domenico, Antonio Lima, Paul Mougel, and Mirco Musolesi. The anatomy of a scientific rumor. *Scientific reports*, 3, 2013.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[eR16] Ahmad Naser eddin and Pedro Ribeiro. Motif-discovery - cytoscape plugin. `http://apps.cytoscape.org/apps/motifdiscovery`, 2016.

[Erc15] Kayhan Erciyes. *Complex Networks: An Algorithmic Perspective*. CRC Press/Taylor & Francis Group, 2015.

[FFHV07] Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 340–351, 2007.

[For96] Scott Fortin. The graph isomorphism problem. Technical report, Technical Report 96-20, University of Alberta, Edomonton, Alberta, Canada, 1996.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

REFERENCES

[FSV01] Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[GK07] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. *Research in Computational Molecular Biology*, pages 92–106, 2007.

[Goo15] Google. Mapreduce for c. `https://github.com/google/mr4c`, 2015.

[Gra16] Martin Grandjean. Connected world: Untangling the air traffic network. `http://www.martingrandjean.ch/connected-world-air-traffic-network/`, 2016.

[HD14] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[How10] Editor Dennis Howe. The free on-line dictionary of computing. `http://www.foldoc.org/`, 2010.

[JMA07] Ruoming Jin, Scott McCallen, and Eivind Almaas. Trend motif: A graph mining approach for analysis of dynamic complex networks. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 541–546. IEEE, 2007.

[KAE+09] Z. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC bioinformatics*, 10(1):318, 2009.

[KFMH+11] L Krumov, C Fretter, M Müller-Hannemann, K Weihe, and M-T Hütt. Motifs in co-authorship networks and their relation to the impact of scientific publications. *The European Physical Journal B*, 84(4):535–540, 2011.

[KIMA04] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[KMP+01] S Kalir, J McClure, K Pabbaraju, C Southward, M Ronen, S Leibler, MG Surette, and U Alon. Ordering genes in a flagella pathway by analysis of expression kinetics from living bacteria. *Science*, 292(5524):2080–2083, 2001.

[Kun16] Jérôme Kunegis. The koblenz network collection. `http://konect.uni-koblenz.de/`, 2016.

[LFS06] Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):360–368, 2006.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[LQLC15] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[MIK+04] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, March 2004.

[ML12] Julian McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In P. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 548–556. 2012.

[MMFDC14] Luis AA Meira, Vinícius R Máximo, Álvaro L Fazenda, and Arlindo F Da Conceição. Acc-motif: accelerated network motif detection. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(5):853–862, 2014.

[MNAR+12] Ali Masoudi-Nejad, Mitra Ansariola, Zahra Razaghi, Ali Salehzadeh-Yazdi, and Sahand Khakabimamaghani. Cytokavosh plugin. `http://apps.cytoscape.org/apps/cytokavosh`, 2012.

REFERENCES

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[MSOI+02] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[New06] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):036104, 2006.

[New10] Mark Newman. Network data sets. `http://www-personal.umich.edu/~{}mejn/netdata/`, 2010.

[NLGC02] Kim Norlen, Gabriel Lucas, Mike Gebbie, and John Chuang. Eva: Extraction, visualization and analysis of the telecommunications and media ownership network. In *Proceedings of International Telecommunications Society 14th Biennial Conference (ITS2002), Seoul Korea*. Citeseer, 2002.

[OSKK05] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E*, 71(6):065103, 2005.

[OSMN09] Saeed Omidi, Falk Schreiber, and Ali Masoudi-Nejad. Moda: an efficient algorithm for network motif discovery in biological networks. *Genes & genetic systems*, 84(5):385–395, 2009.

[Pla13] Todd Plantenga. Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, 73(2):164–175, 2013.

[PR13] Pedro Paredes and Pedro Ribeiro. Towards a faster network-centric subgraph census. In *International Conference on Advances in Social Networks Analysis and Mining*, pages 264–271. IEEE, 2013.

[Pržˇ07] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[RS10] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *ACM Symposium on Applied Computing*, 2010.

[RS14a] Pedro Ribeiro and Fernando Silva. Discovering colored network motifs. In *Proceedings of the 5th International Workshop on Complex Networks (CompleNet)*, March 2014.

[RS14b] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28:337–377, March 2014.

[RSL10a] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Efficient parallel subgraph counting using g-tries. In *IEEE International Conference on Cluster Computing (Cluster)*, pages 1559–1566. IEEE Computer Society Press, September 2010.

[RSL10b] Pedro Manuel Pinto Ribeiro, Fernando MA Silva, and Luís MB Lopes. Parallel calculation of subgraph census in biological networks. In *BIOINFORMATICS*, pages 56–65, 2010.

[RSL12] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Parallel discovery of network motifs. *Journal of Parallel and Distributed Computing*, 72:144–154, 2012.

[ŠB11] Lovro Šubelj and Marko Bajec. Robust network community detection using balanced propagation. *The European Physical Journal B*, 81(3):353–362, 2011.

[SBS+11] D. Shasha, Gary Bader, D. Skripin, A. Pulvirenti, G. Pigola, R. Giugno, and A. Ferro. Netmatch plugin. `http://apps.cytoscape.org/apps/netmatch`, 2011.

[SCBB08] Michael Schatz, Elliott Cooper-Balis, and Adam Bazinet. Parallel network motif finding. *Techinical report, University of Maryland Insitute for Advanced Computer Studies*, 2008.

[SJ15] Saeed Shahrivari and Saeed Jalili. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing*, 97(11):1101–1120, 2015.

[SK04] Olaf Sporns and Rolf Kötter. Motifs in brain networks. *PLoS Biol*, 2(11):e369, 2004.

[SMK93] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. *Optimizing join orders*. Citeseer, 1993.

REFERENCES

[SMO+03] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 2003.

[SOMMA02] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature genetics*, 31(1):64–68, 2002.

[SS04] Falk Schreiber and Henning Schwobbermeyer. Towards motif detection in networks: frequency concepts and flexible search. *Proc. Intl. Wsh. Network Tools and Applications in Biology (NETTAB'04)*, pages 91–102, 2004.

[TMP12] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[Ull76] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[vB12] Lovro Šubelj and Marko Bajec. Software systems through complex networks science: Review, analysis and applications. In *Proceedings of the First International Workshop on Software Mining*, SoftwareMining '12, pages 9–16, New York, NY, USA, 2012. ACM.

[VKK15] Vartika Verma, Paul Park Kwon, and Wooyoung Kim. Iterative hadoop mapreduce-based subgraph enumeration in network motif analysis. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 893–900. IEEE, 2015.

[VS05] Sergi Valverde and Ricard V Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2):026107, 2005.

[Wer06] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions oon Computational Biology and Bioinformatics (TCBB)*, 3(4):347–359, 2006.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[WS11] Christopher Whelan and Kemal Sonmez. Graphletcounter - cytoscape plugin.
`http://apps.cytoscape.org/apps/graphletcounter`, 2011.

[WSTB86] John G White, Eileen Southgate, J Nichol Thomson, and Sydney Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 314(1165):1–340, 1986.

[WTZ+05] Tie Wang, Jeffrey W Touchman, Weiyi Zhang, Edward B Suh, and Guoliang Xue. A parallel algorithm for extracting transcriptional regulatory network motifs. In *Bioinformatics and Bioengineering, 2005. BIBE 2005. Fifth IEEE Symposium on*, pages 193–200. IEEE, 2005.

[YCLH06] Kai-Hsiang Yang, Kun-Yan Chiou, Hahn-Ming Lee, and Jan-Ming Ho. Web appearance disambiguation of personal names based on network motif. In *WEBI*, 2006.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.